SPNP User's Manual
Version 6.0

Contact information:
Professor Kishor S. Trivedi
Center for Advanced Computing and Communication (CACC)
Department of Electrical and Computer Engineering
Duke University
Durham, NC 27708, USA
phone: (919) - 660 - 5269
fax: (919) - 660 - 5293
kst@ee.duke.edu

September, 1999

Preface

The Stochastic Petri Net Package (SPNP) is a versatile modeling tool for performance, dependability and performability analysis of complex systems. Input models based on the theory of stochastic reward nets are solved by efficient and numerically stable algorithms. Steady-state, transient, cumulative transient, time-averaged and up-to-absorption measures can be computed. Parametric sensitivity analysis of these measures is possible. Some degree of logical analysis capabilities are also available in the form of assertion checking and the number and types of markings in the reachability graph. Advanced constructs, such as marking dependent arc multiplicities, guards, arrays of places and transitions, are available. The modeling complexities can be reduced with these advanced constructs. In addition, the expressiveness of the package is enhanced. The most powerful feature of SPNP is the ability to assign reward rates at the net level and subsequently compute the desired measures of the system being modeled. Although no previous knowledge of the C language is necessary to use SPNP, the modeling description language is CSPL, a C-like language.

In the latest version (V6.0), the user can specify non-Markovian SPNs and Fluid Stochastic Petri Nets (FSPNs). Such SPNs are solved using discrete-event simulation rather than by analytic-numberic methods. Several types of simulation methods are available: standard discrete event simulation with independent replicarions or batches, importance splitting techniques (splitting and Restart), importance sampling, regenerative simulation without or with importance sampling, and thinning with independent replications, batches or importance sampling.

Since a graphical user interface, iSPN, is now available, the need for the knowledge of C language further diminishes.

Contents

Preface **ii**

1 Introduction **1**

2 Notation and Terminology **3**

2.1 Petri net and stochastic Petri net 3

2.1.1 Bag 3

2.1.2 Petri net 3

2.1.3 Stochastic Petri net 4

2.1.4 Important feature for SPNP: marking dependence 5

2.2 Non-Markovian SPN 6

2.3 Fluid Stochastic Petri net 6

3 Getting Started With SPNP **7**

3.1 Installation and run 7

3.2 Output files 8

4 The CSPL Language **9**

4.1 Function: options() 9

4.1.1 Functions to set option values 10

4.1.2 Functions to accept runtime inputs 11

4.2 Function: **net()** 11

4.2.1 Functions to define a simple SRN 12

4.2.2 An example: using the power of ANSI C 14

4.2.3 Rate-dependent function 16

4.2.4 Marking-dependent function 16
4.2.5 Specify marking-dependent enabling functions 17
4.2.6 Specify marking-dependent firing rates or firing weights 17
4.2.7 Specify marking-dependent arc cardinalities 18
4.2.8 Define and use parameters 19
4.2.9 Functions to define an FSPN 20
4.2.10 Functions for non-Markovian SPNs 23
4.3 Function: **assert()** 28
4.4 Functions: **ac_init()** and **ac_reach()** 29
4.5 Function: **ac_final()** 29
4.6 A complete example 36

5 Specialized Output Functions **39**

6 Discrete-Event Simulation **43**

6.1 Standard discrete event simulation 44
6.1.1 Current limitations of the simulator 46
6.1.2 Examples 46
6.2 Importance splitting for rare events 46
6.2.1 Example 47
6.3 Importance sampling 47
6.4 Regenerative simulation 49
6.5 Regenerative simulation with importance sampling 49

7 Available Options **50**

7.1 Options for intermediate files 51

7.2 Options for analytic-numeric solution 53
7.3 Options for simulative solution 55
7.4 Miscellaneous options 58

8 Format of the Intermediate Files **61**

8.1 The ".rg" file 61
8.2 The ".mc" file 62
8.3 The ".prb" file 63

9 User guide for iSPN **65**

9.1 Introduction 65
9.1.1 Organization of this guide 65
9.1.2 Conventions used in this chapter 65
9.2 iSPN 65
9.2.1 Why iSPN? 65
9.3 iSPN interface 66
9.4 The Petri net editor 69
9.4.1 File functions 72
9.4.2 Miscellaneous functions 73
9.4.3 Environment control functions 74
9.4.4 Information functions 74
9.4.5 FSPN model 75
9.5 Execution of the model 75
9.6 Viewing output 77
9.6.1 Graphing functions 78
9.6.2 Graph definition functions 80

9.7 Debugging 80
9.7.1 Reachability graph traversal 81
9.8 Browse examples 82
9.9 Help 82
9.10 How to install iSPN in a unix environment 82
9.11 Programming resources 84
9.11.1 Tcl (version tcl7.4) 84
9.11.2 Tk (version tk4.0) 84
9.11.3 Tix (version Tix4.0.4) 85

10 Examples **86**

10.1 Molloy's example 86
10.1.1 Source 86
10.1.2 Description 86
10.1.3 Features 87
10.1.4 SPNP File — *example1.c* 87
10.2 Software Performance Analysis 88
10.2.1 Description 88
10.2.2 Features 89
10.2.3 SPNP File — *example2.c* 89
10.3 $M/M/m/b$ queue 91
10.3.1 Description 91
10.3.2 Features 91
10.3.3 SPNP File — *example3.c* 92
10.4 C.mmp system performability analysis 95

10.4.1 Source 95
10.4.2 Description 95
10.4.3 Features 95
10.4.4 SPNP File — *example4.c* 96
10.5 Database system availability analysis 99
10.5.1 Source 99
10.5.2 Description 99
10.5.3 Features 100
10.5.4 SPNP File — *example5.c* 101
10.6 ATM network under overload 104
10.6.1 Source 104
10.6.2 Description 105
10.6.3 Features 105
10.6.4 SPNP File — *atm.c* 105
10.7 Criticality Importance and Birnbaum Importance 110
10.7.1 Source 110
10.7.2 Description 110
10.7.3 Features 111
10.7.4 SPNP File — *sun.c* 111
10.8 Channel recovery scheme in a cellular network 112
10.8.1 Source 112
10.8.2 Description 112
10.8.3 Features 113
10.8.4 SPNP File — *icupc98.c* 113

10.9 Accurate Model for the BUS in ATM LAN emulation 116
10.9.1 Source 116
10.9.2 Description 116
10.9.3 Features 116
10.9.4 SPNP File — *LAN.c* 116
10.10 Birth-death Model for the BUS in ATM LAN emulation 124
10.10.1 Source 124
10.10.2 Description 124
10.10.3 Features 124
10.10.4 SPNP File — *LA.c* 125
10.11 MMPP Model for the BUS in ATM LAN emulation 130
10.11.1 Source 130
10.11.2 Description 130
10.11.3 Features 131
10.11.4 SPNP File — *LANE.c* 131
10.12 Performance anlysis of Multi-Protocol Label Switching Network 137
10.12.1 Source 137
10.12.2 Description 137
10.12.3 Features 138
10.12.4 SPN model for LSN 138
10.12.5 Files list 138
10.13 Simulation example: reader and writer sharing buffer 153
10.13.1 Source 153
10.13.2 Description 153

10.13.3 Features 153
10.13.4 SPNP File — *readwrite.c* 154
10.14 Hybrid System: reactor temperature control system 155
10.14.1 Source 155
10.14.2 Description 155
10.14.3 Features 155
10.14.4 SPNP File — *reactor.c* 156
10.15 Dual tank example 157
10.15.1 Source 157
10.15.2 Description 157
10.15.3 Features 157
10.15.4 SPNP File — *splitting.c* 157
10.16 Equivalent failure rate and repair rate computation in hierarchical model . 160
10.16.1 Source 160
10.16.2 Description 160
10.16.3 Features 161
10.16.4 SPNP Files for Module level Markov chain model 161
10.16.5 SPNP File for System level SPN model — *gsb.c* 171
10.17 Analysis of Phased-Mission Systems (PMS) with DSPN 179
10.17.1 Source 179
10.17.2 Description 180
10.17.3 SPNP File — *pms.c* 180
10.17.4 Configuration File — *pms.cfg* 189
10.17.5 Shell File — *t.csh* 190

10.18 Extensions to SPNP 190
10.18.1 Fixed point iteration 190
10.18.2 Initial probability reload 191

A Differences Between last versions of SPNP **197**

A.1 Command Differences Between SPNP Version 4 and Version 5 197

A.2 Command Differences Between SPNP Version 5 and Version 6 197

B SPNP Applications **198**

Bibliography **203**

Index **207**

Chapter 1

Introduction

The Stochastic Petri Net Package (SPNP) is a versatile modeling tool for solution of Stochastic Petri Net (SPN) models. The SPN models are described in the input language for SPNP called CSPL (C-based SPN Language). The CSPL is an extension of the C programming language [13] with additional constructs which facilitate easy description of SPN models. The full power and generality of C is available, but a working knowledge of C is sufficient to use SPNP effectively.

The SPN models specified to SPNP are actually "SPN Reward Models" or Stochastic Reward Nets (SRNs) [6, 7] which are based on the "Markov Reward Models" (MRM) paradigm [15, 28]. Fig. 1.1 shows the taxonomy of SRN models and the solution method implemented in SPNP. Markov Reward Model provides a powerful modeling environment for:

- Dependability (Reliability, Availability, Safety) analysis.
- Performance analysis.
- Performability modeling.

A number of important Petri net constructs such as marking dependency, variable cardinality arc and enabling functions (or guards) [6] facilitate the construction of models for complex systems. The package also allows logical analysis on the Petri net whereby any general assertions defined on the Petri net are checked for each marking of the net. The SRN may be solved to obtain either steady-state metrics or transient metrics. The package allows the specification of custom measures although a standard set of measures are available. The measures are defined in terms of reward rates associated with the markings of the SRN. Parametric sensitivity analysis allows the user to evaluate the effect of changes in an input parameter on the output measures. This is useful in system optimization and bottleneck analysis. The discrete-event simulation facility may be used to solve large Markovian SPNs and non-Markovian SPNs. Basic Fluid Stochastic Petri Nets (FSPNs) [8, 14, 32], whose analytic-numerical solution is not supported in the current version, can also be simulated, and results are reported in a way similar to those for the numerical solution. Note that if the SPN is solved using simulation, the reachability graph will not be generated.

This manual describes SPNP Version 6.01, running under the UNIX system on a variety of platforms (VAX, Sun 4, 5 and Ultra, Convex, Gould, NeXT, CRAY), AIX system

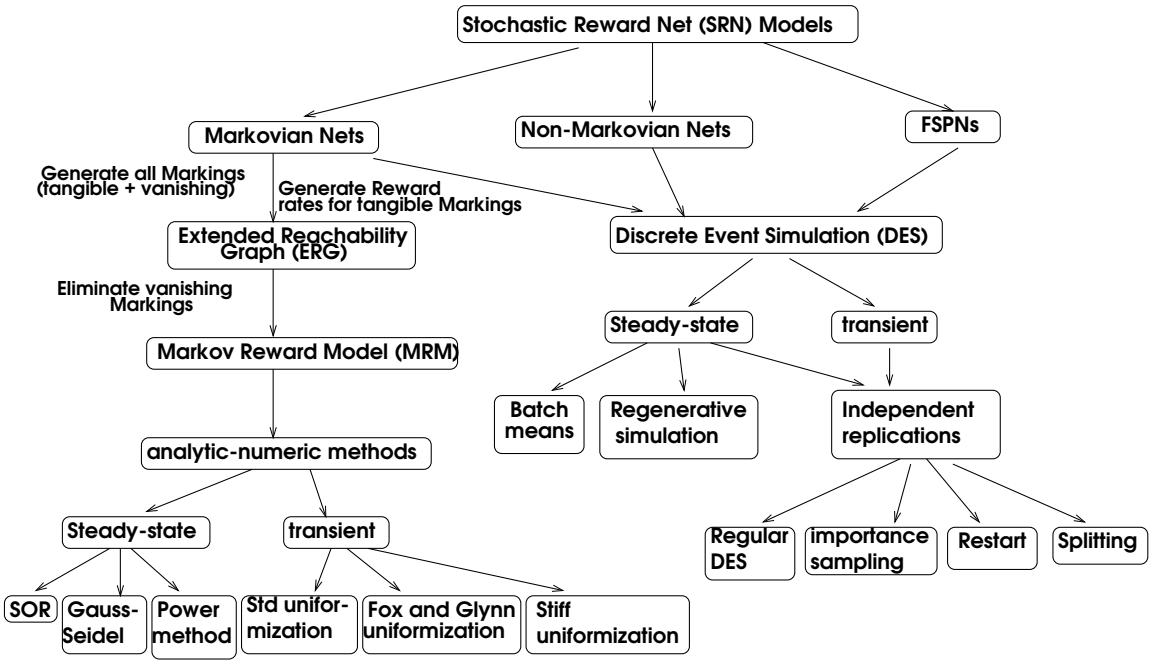


Figure 1.1: Taxonomy of SRN models and analysis methods

(RS/6000), Linux, VMS system (VAX). The description will apply mainly to UNIX-based systems.

A basic knowledge of the stochastic Petri net (SPN) formalism and Markov chains is assumed. The reader should consult [24, 25] if unfamiliar with some Petri net (PN) concepts. The Markovian SPN model we adopt is best described in [5, 6, 7], but it may be useful to consult [1]. FSPN model we assume is best described in [8] while the non-Markovian SPN model is best described in [11], but enhanced with the list of distributions of this manual. For a reference on the C language, see [13]. For further information on Markov chains, performance modeling, and reliability modeling see [3, 31] while for performability modeling see [27, 28, 33]. Markov and Markov reward model solution techniques are surveyed in [26]. Sensitivity analysis of Markov and Markov reward models is discussed in [2] and the sensitivity analysis of SRN models is discussed in [21]. Several papers have appeared in the literature where SPNP was used, some of these papers are listed in Appendix B. A 6-hour long VHS tape for a course on "Putting Stochastic Petri Nets to Work," by K. S. Trivedi and G. Ciardo can be ordered from USC-ITV by calling +1-213-740-0119.

Although model hierarchies are not built into SPNP, hierarchical SRN models can be exercised using a UNIX shell file and submodels can communicate information via files that can be declared and opened inside individual SPNP submodel input files. Examples of papers using model hierarchies and fixed-point iteration include [4, 10, 16, 19, 20, 22, 30, 17, 23].

Chapter 2

Notation and Terminology

We write predefined CSPL types, constants, and functions in **boldface**, while we use *italic* for user-defined quantities. In the examples, however, we simply use a `fixed-pitch font` to show actual CSPL code. For readability, a place `p_1` or a parameter `alpha` in a CSPL fragment are written as p_1 and α in the textual discussion.

2.1 Petri net and stochastic Petri net

2.1.1 Bag

The concept of bag will be used in the following, so we describe it here. For a complete definition and examples see [25]. The concept of bag extends the one of set. If x is an element of the set S, then $S \cup \{x\} = S$, but there are cases where it is important to count the occurrences of x in S. A bag represents this by allowing repeated occurrences of the same element, or, in other words, by attaching a positive integer count to each element of a set. So, for example, $\{x, x, y, z\} - \{x, y\} = \{x, z\}$.

2.1.2 Petri net

A Petri net is a directed graph whose nodes are partitioned into two sets, *places* and *transitions*. *Arcs* can only connect a place to a transition (input arcs), or a transition to a place (output arcs). A *multiplicity* (positive integer) may be attached to each arc, which is then called a multiple arc. Intuitively, a multiple arc with multiplicity k can be thought of as k arcs having the same source and destination.

The input (output) bag for a transition is the bag constituted by the input (output) arcs, considered with their multiplicity. Each place may contain any number of tokens. All the tokens are indistinguishable. A marking is a bag representing the configuration of tokens in the places of the Petri net. It is also called the state of the Petri net.

Regarding the evolution of the Petri net, the following terms are fundamental. A transition is *enabled* if its input bag is a subbag of the (current) marking. When a transition is enabled, it can *fire*, leading the Petri net into a possibly different marking, obtained by

subtracting its input bag from and adding its output bag to the current marking. A firing sequence is a sequence of transition firings. A marking is *reachable* if it is obtained by a firing sequence starting in the initial marking. The *reachability set* (graph) is the set (graph) of all the reachable markings (connected by arcs labeled with the transition firings). A set S of transitions enabled in a marking m is a conflicting transition set if the contemporary firing of all the transitions of S is impossible in m, or, in other words, if the sum of the input bags of the transitions in S is not a subbag of m.

2.1.3 Stochastic Petri net

The stochastic Petri net model is obtained from the Petri net model by associating a probability distribution function to the firing time of each transition. Additional constructs are often present as well. In the Generalized Stochastic Petri net model [1], only two distribution types are allowed: exponential and deterministic with value 0. Transitions with an associated exponential distribution are said to be *timed*; transitions with zero time distribution are said to be *immediate*. In the Extended Stochastic Petri net model [12], the transitions are classified in a similar way, but an arbitrary distribution can be associated to each timed transition. If two or more conflicting transitions should fire at the same moment (this event has a 0 probability if the distribution is continuous), a probability mass function must specify the probability that a subset of transitions will actually fire. The version of SPN that we deal with is known as the stochastic reward net (SRN).

The parameters of an exponential or general distribution are said to be *marking dependent* if they can be different in each marking. This is allowed by our definition; we will show how to describe this marking dependency.

Additional constructs are used to selectively disable a transition in a marking which would otherwise enable it. A *priority* is associated with each transition. If S is the set of transitions enabled in a marking and if the transition with the highest priority among them is k, then any transition in S with priority lower than that of transition k will be disabled. To avoid theoretical difficulties, timed and immediate transitions cannot have the same priority. Another way to disable a transition is the *inhibitor arc*. An inhibitor arc from place p to transition t with multiplicity m will disable t in any marking where p contains at least m tokens. If these two constructs are not sufficient to describe a particular mechanism, the marking dependent enabling function (also called a *guard*) with each transition can be used: if this function evaluates to 0 in a marking, then the transition is disabled. If we ignore timing, we can imagine an ordinary Petri net as a Stochastic Petri net where all transitions have the same priority, where no inhibitor arcs are present, and where the enabling functions are identically equal to 1.

A marking is *tangible* if it enables no immediate transition; it is called a *vanishing* marking otherwise. A marking which does not enable any transition is *absorbing*, hence it

is tangible by definition. A (maximal) set of vanishing markings that are mutually reachable by immediate transition firings is called a *loop* (of vanishing markings). A loop is said to be absorbing if no marking in it reaches a marking outside the loop; otherwise the loop is transient. An absorbing loop is considered an error. Transient loops are not a problem, their interpretation is clear, and they are easily and correctly managed by the package, but, if you know that your SRN should not contain a transient loop, you should look for them, since they could be the manifestation of a modeling error. The reachability graph contains an arc for each different transition enabled in each marking: in particular, self transitions (with equal input bags and output bags) are allowed by the definition of the model, so arcs with coinciding source and destination may be present in the reachability graph. These arcs are ignored during the solution steps.

2.1.4 Important feature for SPNP: marking dependence

An important feature of the SRN model is the *marking dependent* arc multiplicities, enabling functions firing probabilities and firing rates. SPNP is the software package that supports the specification and the solving of the SRN models.

Arcs can have a multiplicity which is not constant, but rather it is a function of the marking. This possibility was defined because it may allow substantial reductions in the size of the reachability graph; it also may allow to model, in a compact way, behaviors that would otherwise require complex subnets. A typical example is the case where all the tokens from place p must be moved to place q when transition t fires. An input arc from p to t and an output arc from t to q, both with marking dependent multiplicity equal to the number of tokens in place p are enough to model this behavior. Without this construct, the reachability graph would contain all the intermediate arcs and markings corresponding to the movement of tokens, one by one. Perhaps even more importantly, if t is timed, the stochastic behavior will not be the same, unless the SRN explicitly models this "flushing" of tokens with an additional immediate transition and possibly some control places. Some words of caution must be said on this construct. First, it should be used only when really needed, because it may make the SRN harder to understand and it requires slightly more computation than a standard or multiple arc. This is because the input and output bags for a transition in a marking are computed by evaluating the marking dependent functions for the arc multiplicities, if any, before firing the transition (this is why the output arc from t to q will put the correct amount of tokens in q). This might give rise to unintuitive or unforeseen behaviors; for example, in the flushing of tokens just described, transition t is enabled in any marking, even when place p is empty, unless (1) other input arcs are defined for t, (2) an enabling function is used to explicitly disable t when p is empty and possibly in other cases as well, or (3) the marking dependent arc multiplicity function for the arc from p to t returns a positive value when p is empty (this is the most efficient solution if the goal is to enable t only when there are one or more tokens to be flushed in p).

At times, inhibitor arcs or transition priorities can specify a given behavior only through awkward subsets that only obfuscate the actual logic of model. In these cases, the definition of a marking-dependent enabling function (or a guard) is probably a better choice.

Marking-dependent functions can be used to specify the *firing rate* of a timed transition or the firing probability of an immediate transition.

2.2 Non-Markovian SPN

A major restriction to the SRN model described previously is that the transitions are either immediate or exponentially distributed. Therefore some other distributions have been introduced in SPNP in the last a few years. When a transition has a general distribution of firing time, the SPN is called a *Non-Markovian SPN*. This complicates the numeric-analytic or simulative method used to solve the model. At the present time, the transitions can have constant, uniform, geometric, Weibull, normal, lognormal, Erlang, even the hyperexponential distributions. The complete list of distributions implemented and being implemented in SPNP can be found at the beginning of Chapter 6. Here again, the distribution parameters can be marking dependent.

2.3 Fluid Stochastic Petri net

SPNP also can simulate and solve FSPNs. FSPN is an fluid extension to SRN, parrallel to the fluid queueing network extension to ordinary queueing network. The main difference between SRNs and FSPNs is that, in addition to ordinary (discrete) places, a FSPN can also have fluid places containing a (real) fluid level between zero and an upper bound, possibly infinity. An input or output arc connected to a fluid place removes or adds fluid continuously while the transitions is enabled, as long as the level remains between zero and the upper bound.

FSPNs have the potential to facilitate the computation of two kinds of important problems: One is the ordinary SPN with a huge amount of tokens in some places, which can now be approximated by fluid places; the other is for analysing hybrid systems with a continuous deterministic part and a discrete stochastic part, which is hard to deal with by other hybrid system analyzation tools. FSPN can be solved either with numeric- analytic method or with simulative method. At the present time, only the simulative method is implemented in SPNP. For further information on FSPNs, see [8, 14, 32].

Chapter 3

Getting Started With SPNP

3.1 Installation and run

The package is composed of several C files. The SRN to be studied must be described in a CSPL (C-based Stochastic Petri Net Language) file, which is a C file specifying the structure of the SRN and the desired outputs, by means of predefined functions. The CSPL file is compiled, linked to the other files constituting the package (usually kept compiled), and run, by typing

```
make -f /PATH_TO_SPNP/spnp/obj/Makerun SRN=filename
```

where $filename$ is your CSPL file, without the C extension, and PATH_TO_SPNP represents the UNIX path to the package on your installation.

It is possible to check the CSPL file for certain errors, by typing

```
make -f /PATH_TO_SPNP/spnp/obj/Makerun lint SPN=filename
```

If inconsistencies exist between the definition of the predefined functions and their usage, they will be discovered.

Note: To save typing, you can define the aliases

```
alias spnp "make -f /PATH_TO_SPNP/spnp/obj/Makerun SPN=\!^"
alias spnpcheck "make -f /PATH_TO_SPNP/spnp/obj/Makerun lint SPN=\!^"
```

(assuming that "!" is your history character) and type, regardless of your current directory,

```
spnp filename
spnpcheck filename
```

3.2 Output files

The intermediate files generated by the package and the final results will be in the same directory as where $filename$**.c** is (and where you issued the command). Files have different extensions, according to the kind of information they carry. If your CSPL file is named **test.c**, then the following files will be generated:

- **test.o**: obtained when compiling **test.c**.
- **test.spn**: executable file obtained by linking the package object files together with **test.o**.
- **test.rg**: containing the reachability graph information: composition of each marking, description of the transition firings between them, etc.
- **test.mc**: containing the (numerical) CTMC/DTMC corresponding to the SRN
- **test-**$parmname$**.mc**: containing the (numerical) derivative of CTMC with respect to parameter $parmname$ (one file is generated for each parameter defined in **test.c**.
- **test.prb**: containing the (numerical) results of the analysis of the underlying CTMC: the transient or steady-state probabilities for each tangible marking, the cumulative sojourn times in transient states up to the solution time, and the derivatives (with respect to defined parameters) of the aforementioned measures.
- **test.prbdtmc**: containing the (numerical) results of the embedded DTMC.
- **test.out**: containing the requested output (according to what is specified in **test.c** using the provided functions).
- **test.log**: contains all the output messages produced by the package during model solution.
- **test.dot**: contains a description of the Petri net in the *dot* graph language.

Chapter 4

The CSPL Language

The CSPL languange can be defined as a system for describing, handling and solving Stochastic Reward Nets (SRNs). The syntax and the semantics of CSPL are based on the ANSI C language. Moreover, a correct CSPL file is a correct ANSI C file too. What distinguishes CSPL from ANSI C is a set of predefined functions for the specification and solving of SRNs. Familiarity with ANSI C languange will be a great advantage for a user to exploit most of the features of the CSPL language. However, it should be without difficulty for a user to be able to define and solve his/her SRNs under study with the help of this manual, especially the examples provided within it.

Any legal ANSI C construct is allowed in the CSPL language, as needed. User-defined variables and functions can be used in a CSPL file. In particular, all the standard C library functions, such as **fprintf()**, **fscanf()**, **log()**, **exp()**, etc., can be called in CSPL, if necessary.

A CSPL file must contain the following five basic functions:

- **options()**,
- **net()**,
- **assert()**,
- **ac_init()**,
- **ac_reach()**,
- **ac_final()**.

Each function listed is designed to carry out one (or several) certain task(s) by calling some relevant subroutines (functions). These tasks are related to the definition and solving processes of stochastic reward nets. In the following sections we will discuss the function above and the other related subroutines in detail. Simple examples will be given to illustrate their usage.

4.1 Function: options()

void options(void);

A valid CSPL file must contain the **options()** function. Function **options()** calls the following subroutins:

- **iopt()** ,
- **fopt()** ,
- **input()** , and
- **finput()** ,

to set options which will affect the way of describing and solving SRN. A detailed list of these options can be found in Chapter 7, Availabe Options.

4.1.1 Functions to set option values

void iopt(IOP_TYPE *option***, int** *val***);**

void fopt(FOP_TYPE *option***, double** *val***);**

Functions **iopt()** and **fopt()** set option indexed as *option* with value *v*. There are two kinds of options, integer options and float options (more precisely, options with double precision floating point values). A user is advised to be aware of the difference and use the correct type while setting options.

Example:

```
void options() {
   ......
   iopt(IOP_PR_MARK_ORDER, VAL_LEXICAL);
   ......
   fopt(FOP_PRECISION, 1e-8);
   ......
}
```

The above code causes the markings to be printed in lexical order instead of the default canonical order and the precision for a numerical solution is set to $1e-8$.

4.1.2 Functions to accept runtime inputs

int input(char *msg);

double finput(char *msg);

Functions **input()** and **finput()** accept input from standard input during run time while function **option()** is called. Function **input()** accepts an integer value while function **finput()** accepts a floating point value.

Example:

```
int num_customers;
options() {
  ...
  num_customers = input("Number of customers");
  ...
}
```

The above CSPL code causes the following message specified by the string parameter *msg*,

```
INPUT "Number of customers" (int) >
```

to be displayed on the screen (more precisely, on the **stderr** stream), then SPNP waits for the user to type a value. The input value is accepted by SPNP and is printed in the "**.out**" file, together with the string msg. This is useful to recall the set of values input to a particular CSPL file to generate the current output. The returned input value can be assigned to any variable declared by the user and can be used in the rest of the CSPL file. (Note: be sure to use the variable after the desired value is assigned.)

4.2 Function: net()

void net(void);

A valid CSPL file must contain function **net()**. Function **net()** calls a set of functions to define an SRN. The following description starts with a small set of functions to define a simple SRN followed by an example to construct a relatively complex SRN.

4.2.1 Functions to define a simple SRN

The following funcitons are sufficient to define a simple SRN without extended features of marking-dependent enabling, rates/weights and cardinalities, etc:

- **place()** and **init()**

  The function

  void place(char *p);

  defines a place with name *p*. A name is legal if: (1) its length is between 1 and **MAX_NAME_LENGTH**, as defined in the file **const.h** [1] (typically 20); (2) it is composed of the characters {**0..9, a..z, A..Z, _**} only; (3) the first character is in {**a..z,A..Z**}. All names must be distinct, that is, it is an error to have two places, two transitions, or a place and a transition having the same name.

  The function

  void init(char *p, int n);

  defines the initial number of tokens in Place p to be n. By default, places are otherwise initially empty.

- **imm()**

  The timed transitions are automatically defined when their rates (or distributions) are defined (see the following). However, immediate transitions must be defined by

  void imm(char *t);

- **rateval()** and **probval()**

  The functions

  void rateval(char *t, double val);

  void probval(char *t, double val);

  define the firing rate of timed transition t and the firing weight (unnormalized probability) of immediate transition t, as a constant value val. Function **rateval** (or its more general versions illustrated in the following sections) implicitly defines the timed transition must and be defined for each transition. Function **probval** (or its more general versions illustrated in the following sections) needs to be called only if the firing weight is different with the default value 1.0 coming with the definition of the immediate transition by **imm**.

[1] The definitions in the files **const.h** and **type.h** are always provided with the distribution of the package. Users are not encouraged to change them especially when the complete souce files are not available. Changes in these files may incur inconsistency and may cause compiling failure.

- **priority()**

  The function

  void priority(char *_t_, int *prio***);**

  defines the priority for Transition *t* to be *prio*. By default transitions have the lowest priority (0).

- **policy()**

  The function

  void policy(char *_t_, int *pol***);**

  defines the resampling policy for transition *t* to be *pol* when the (enabled) transition is disabled by the firing of a competitive transition and later becomes enabled again.[2] Three different policies are implemented:

  – **PRI**(Preemptive Repeat Identical): the interrusted job is repeated with an identical firing time.

  – **PRD**(Preemptive Repeat Different): the interrupted job is repeated with a resampled random time.

  – **PRR**(PReemptive Resume): an interrupted job continues with the old remaining firing time.

  By default transitions have policy **PRD**.

- **affected()**

  The function

  void affected(char *_s_, char *_t_, int *pol***)**

  defines the effect on the firing time of transition *s*, if it remains enabled after some other transition *t* fires. *pol* should be **PRI**, **PRS** or **PRD**. The default is set to be **PRS** for non-memoryless distribution of *s* with **PRD** for exponential distribution.

- **iarc()**, **oarc()**, **harc()**, **miarc()**, **moarc()**, and **mharc()**

  The functions

  void iarc(char *_t_, char *_p_);

  void oarc(char *_t_, char *_p_);

  void harc(char *_t_, char *_p_);

[2] This function is used mainly in simulation. Please refer to the example of reader and writer sharing buffer in Sec. 10.13.

void miarc(char **t*, char **p*, int *mult*);

void moarc(char **t*, char **p*, int *mult*);

void mharc(char **t*, char **p*, int $mult$);

define, respectively, an *input arc* from Place p to Transition t, an *output arc* from Transition t to Place p, or an *inhibitor arc* from Place p to Transition t with multiplicity one or $mult$ (a positive **int**).

- **halting_condition()**

  The function

  void halting_condition(int (gfunc*)())**

  defines the halting condition *gfunc* for the SPN. When this function evaluates to zero, the marking is considered absorbing.

4.2.2 An example: using the power of ANSI C

So far, we have introduced convenient functions to get runtime input and basic functions to define an SRN. The purpose of this section is to illustrate how to take advantage of the flexibility of ANSI C to facilitate the definition of a complex system with those simple CSPL functions we have so far introduced. We show an example (Figure 4.1) which defines a subnet of Erlang distribution.

The portion of CSPL file in Figure 4.1 allows the run-time definition of the number of stages in a subnet corresponding to an Erlang distribution. The size of the arrays of "ph" Places and "th" transitions, is determined respectively as $max + 1$ and max at run-time using the predefined **input** function.

You will see a message on the terminal

```
Please type "number of phases" (int) >
```

and the variable max will be set to the number you type, say 6. Then seven places with names ph_0, ..., ph_6 will be defined, with no tokens in them initially. Six transitions will be defined, with names th_1, ..., th_6. Finally, a sequence of both input and output arcs (ph_0,th_1), (th_1,ph_1), ..., (th_6,ph_6) will be defined as well.

```
int   max;

void options() {
     ...
     max = input("number of phases");
     }

void net() {

     int   i;

     char  auxplace[20],auxtrans[20];

     ...

     sprintf(auxplace,"ph_0");
     place(auxplace);

     for (i = 1; i <= max; i++) {
         sprintf(auxtrans,"th_%d",i);
         sprintf(auxplace, "ph_%d",i);

         place(auxplace);
         trans(auxtrans);

         rateval(auxtrans, 2.0);
         iarc(auxtrans, auxplace);
         oarc(auxtrans,auxplace);
         }

 }
```

Figure 4.1: An Erlang subnet

4.2.3 Rate-dependent function

A rate-dependent function is a C function which makes call(s) to functions referring to SRN entity(entities) and returns either an **integer** value or a **double** value. Rate-dependent functions are used to specify for instance output measures which will be described in the following sections.

The following function, defined only for exponential distributions, can be used in marking-dependent functions:

double rate(char *_tr_)

which returns the rate of transition *tr* in the current marking if it is enabled in the current marking and 0 otherwise.[3]

4.2.4 Marking-dependent function

A marking-dependent function is a C function which makes call(s) to functions referring to SRN entity(entities) and returns either an **integer** value or a **double** value. Marking-dependent functions are used to specify marking-dependent firing rates, firing weight, arc cardinalities, and output measures, which will be described in the following sections.

The following two functions can be used in marking-dependent functions:

- **mark()**

  The function

  int mark(char *p);

  returns the number of tokens in Place p.

- **enabled()**

  The function

  int enabled(char *t);

  returns 1 if Transition t is enabled in the current marking; 0, otherwise. [4]

The following code defines a marking-dependent function **f()**.

[3] Refer to Examples 10.1, 10.3, 10.4 and 10.8.

[4] Refer to Examples 10.3 and 10.11.

```
double    f() {
  return(mark("p_1") * mark("p_2") * 7.3);
}
```

This function **f()** returns a value of 7.3 times the product of the number of tokens in Places p_1 and p_2.

4.2.5 Specify marking-dependent enabling functions

At times, inhibitor arcs or transition priorities can specify a given behavior only through awkward subnets that only obfuscate the actual logic of the model. In these cases, the definition of a marking-dependent enabling function (or guard) is probably a better choice.

The function

void guard(char *t, int *$func$());

defines the enabling function of Transition t to be the marking-dependent function $func$, instead of the default, the constant function returning 1 in all markings.

For example, if we defined a function,

```
int    gt() {
  return(mark("p_1") > 2 * mark("p_2"));
}
```

we can then use it in

```
guard("t_1",gt);
```

to guarantee that t_1 is disabled when p_1 does not contain more than twice as many tokens as p_2 does.

4.2.6 Specify marking-dependent firing rates or firing weights

Occasionally it is useful to define the firing rate or firing weight of a transition as a marking-dependent quantity. The simplest way is to define it as a quantity proportional to the

number of tokens in a place.[5]

The functions

void ratedep(char *t, double *val***, char *p);**

void probdep(char *t, double *val***, char *p);**

define the firing rate or firing weight of Transition *t* to be *val* times the number of tokens in Place *p*. (Note: it is an error for a firing rate or firing weight to be evaluated to zero in a marking where the transition is enabled. Hence this method can not be used when Place *p* is not an input place for *t* and can become empty.)

In many other cases, though, a more general type of marking dependency is required. This is achieved by defining a marking-dependent function of type **double**. The functions

void ratefun(char *t, double (*$func$());

void probfun(char *t, double (*$func$());

define the firing rate or firing weight of Transition *t* to be the value of marking-dependent function $func$, evaluated in the current marking.

4.2.7 Specify marking-dependent arc cardinalities

The functions

void viarc(char *t, char *p, int *$func$());

void voarc(char *t, char *p, int *$func$());

void vharc(char *t, char *p, int *$func$());

define, respectively, an input arc from Place *p* to Transition *t*, an output arc from Transition *t* to Place *p*, or an inhibitor arc from Place *p* to Transition *t* with multiplicity given by the marking-dependent function $func$. [6]

We choose to define **vharc** for completeness, but it is usually more efficient to use an guard function instead.

[5] These functions are used extensively in Example 10.10 and 10.16.

[6] Refer to Examples 10.4, 10.6, 10.12, 10.13, and 10.15 for their usage.

4.2.8 Define and use parameters

To perform sensitivity analysis, the rate and weights of one or more transitions must be defined as a function of a **double** parameter.[7] SPNP will then be able to compute the derivatives of the requested output measures with respect to this parameter. This requires to define parameters, and to connect them to the rate or weights of transitions.

The function

void parm(char *x);

defines the existence of a parameter x. The usual naming conventions for places and transitions apply to parameters as well. For implementation reasons, the allowable number of defined parameters is limited to **MAX_PARMS**, defined in **const.h**.

The function

void bind(char *x, double val**);**

sets the value of parameter x to val, until the next call. This function can also be called, multiple times, in function **ac_final**, to bind x to different values, and compute the corresponding measures.

The function

void useparm(char *t, char *x);

associates the rate or weight of Transition t to the parameter x. This function can be called multiple times for a single transition, which can then have, in full generality, a rate of the form

$$\lambda(m) \cdot \prod_{i=1}^{n} x_i^{k_i},$$

where $\lambda(m)$ is the value of a marking-dependent function λ evaluated in marking m, n is the number of different parameters associated to t, and k_i is the number of times the i-th parameter, x_i, has been associated with t.

A parameter may provide itself a value for the rate or weight of a transition. To do so, we begin by defining its rate or weight to be one (it was what procedures **ratenoval** and **probnoval** did in SPNP 5.0; they have been removed as they are equivalent to defining the rate or probability to be one).

For example,

[7] Refer to *sensi.c* in the example CSPL files.

```
parm("alpha");
parm("beta");
parm("gamma");

rateval("t_1",1.0);
useparm("t_1", "alpha");

ratedep("t_2", 10.0, "p");
useparm("t_2", "beta");
useparm("t_2", "beta");
useparm("t_2", "beta");
probval("t_3",1.0);
useparm("t_3", "gamma");
```

defines three parameters, α, β, and γ, and timed transition t_1 with rate α, timed transition t_2 with a marking-dependent rate $10.0 \cdot \mathbf{mark}(p) \cdot \beta^3$, and immediate transition t_3 with weight γ. Before solving the model, the parameters are assigned with numeric values by calling the **bind** function, where the name of the parameter and its value are given.

For example, in

```
int N;
......

bind("alpha",1.0/N);
bind("beta",2);
bind("gamma",0.75);
```

α, β and γ are assigned a values N^{-1} (where N may be set by user input), 2.0, and 0.75, respectively.

4.2.9 Functions to define an FSPN

[8]

- **fplace()**

  The function

  ```
  void fplace(char *$n$);
  ```

[8] Refer to Sec. 10.14 and Sec. 10.15 for examples.

defines a a fluid place, or a transition) with name n. The same restrictions as for place and transition names apply.

- **fbound()** The function

  ```
  void fbound(char *$p$, double $b$);
  ```

  defines the upper bound on the fluid level of fluid place p to be b. The lower bound is always 0.

- **finit()**

  The function

  ```
  void finit(char *$p$, double $l$);
  ```

  defines the initial fluid level in fluid place p to be l, instead of the default 0.

- **fbreak()**

  The function

  void fbreak(char *p, double l);

  defines a threshold in fluid place p to be l. Several threshold may be defined in the same place by calling **fbreak()** for multiple times. This function allows to change parameters after level l is reached at place p.

- **inf()**

  The function

  void inf(char *t);

  declares a transition defined *only* by fluid flow (when enabled).

- The functions

  void fiarc(char *t, char *p);

  void foarc(char *t, char *p);

  void fmiarc(char *t, char *p, double $mult$);

  void fmoarc(char *t, char *p, double $mult$);

  void fviarc(char *t, char *p, double (*$func$)());

  void fvoarc(char *t, char *p, double (*$func$)());

define an input arc from fluid place p to transition t or a fluid output arc from transition t to fluid place p, with fluid flow rate given by 1, the constant $mult$ or the marking-dependent function $func$, respectively.

- The functions

  void fliarc(char *t, char *p, double (*f)(), double (*g)());

  void floarc(char *t, char *p, double (*f)(), double (*g)());

  define an input arc from fluid place p to transition t or a fluid output arc from transition t to fluid place p, with linear fluid flow given by the marking-dependent functions f and g, i.e., in place p the differential equation when discrete marking is m is, with respect to time τ,

  $$\frac{dx_p(\tau)}{d\tau} = f(m)x_p(\tau) + g(m).$$

- In the same way, the functions

  void diarc(char *t, char *p);

  void doarc(char *t, char *p);

  void dharc(char *t, char *p);

  void dmiarc(char *t, char *p, double $mult$);

  void dmoarc(char *t, char *p, double $mult$);

  void dmharc(char *t, char *p, double $mult$);

  void dviarc(char *t, char *p, double (*$func$)());

  void dvoarc(char *t, char *p, double (*$func$)());

  void dvharc(char *t, char *p, double (*$func$)());

  define an input arc from fluid place p to transition t or a fluid output arc from transition t to fluid place p, or an inhibitor arc from place p to transtion t with fluid impulse (or level in the inhibitor case) given by 1, the constant $mult$ or the marking-dependent function $func$, respectively.

- An additional marking-dependent function is available to inquire about the state of a fluid place:

  double fmark(char *p);

  returns the fluid level in fluid place p.

- Finally, the function

int fcondition(char *fpl, char *rel, double val)

return **TRUE** if **fmark(**fpl**)** satisfies the relation with val, **FALSE** otherwise. Here, **rel** must be one of **F_EQ** (equals), **F_NQ** (non-equals), **F_GT** (greater than), **F_LT** (less than), **F_GE** (greater or equal), **F_LQ** (less or equal), a relational operator. This may be useful for user-defined function and guards.

4.2.10 Functions for non-Markovian SPNs

[9]

As an alternative to the functions used to define a timed or immediate transition, the following functions can be used to define timed transitions with non-exponential distribution of the firing time. These can appear only in models solved by discrete-event simulation.

- Functions

  void detval(char *t, double val);

  void detdep(char *t, double val, char *p);

  void detfun(char *t, double (*fun)());

  define a transition t with constant firing time given respectively by val, the constant val times the number of tokens in place p, and the marking-dependent function fun.

- Functions

  void unifval(char *t, double $lower$, double $upper$);

  void unifdep(char *t, double $lower$, double $upper$, char *p);

  void uniffun(char *t, double (*f)(), double (*g)());

  define a transition t with firing time uniformly distributed in the interval with lower and upper bounds given respectively by $lower$ and $upper$, $lower$ times the number of tokens in place p and $upper$ times the number of tokens in place p, or the marking-dependent function f and g.

- Functions

  void geomval(char *t, double $val1$, double $val2$);

  void geomdep(char *t, double $val1$, double $val2$, char *p);

  void geomfun(char *t, double (*f)(), double (*g)());

[9] See Sec. 10.13 for example.

define a transition t with geometrically distributed firing time with respective parameter $val1$, $val1$ times the number of tokens in p and marking-dependent function f, and with respective time step given by $val2$, $val2$ times the number of tokens in place p, or the marking-dependent function g. The probability to have value $n \times val2$, $forall n \geq 1$, is then $val1(1 - val1)^{n-1}$.

- Functions

  void weibval(char *t, double $val1$**, double** $val2$**);**

  void weibdep(char *t, double $val1$**, double** $val2$**, char *p);**

  void weibfun(char *t, double (*f)(), double (*g)());

  define a transition t with Weibull distributed firing time with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, or the marking-dependent function f and g. The density is then is then $\forall x \geq 0$, $val1 \times val2 x^{val2-1} e^{-val1 * x^{val2}}$.

- Functions

  void normval(char *t, double $val1$**, double** $val2$**);**

  void normdep(char *t, double $val1$**, double** $val2$**, char *p);**

  void normfun(char *t, double (*f)(), double (*g)());

  define a transition t with a normally distributed firing time with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, or the marking-dependent function f and g. This normal distribution is truncated by considering only non-negative values. Parameters $val1$ and $val2$ are the expectation and variance of the non-truncated distribution.

- Functions

  void lognval(char *t, double $val1$**, double** $val2$**);**

  void logndep(char *t, double $val1$**, double** $val2$**, char *p);**

  void lognfun(char *t, double (*f)(), double (*g)());

  define a transition t with lognormally distributed firing time with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, or the marking-dependent function f and g. The density is then, $\forall x > 0$,

  $$\frac{1}{\sqrt{2\pi val2}x} e^{-(\ln x - val1)^2 / 2val2}.$$

- Functions

void gamval(char *t, double $val1$, double $val2$);

void gamdep(char *t, double $val1$, double $val2$, char *p);

void gamfun(char *t, double (*f)(), double (*g)());

define a transition t with gamma distribution with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, or the marking-dependent function f and g. The density is then, $\forall x > 0$,

$$\frac{1}{\Gamma(val1)} val2^{val1} x^{val1-1} e^{-val2 \times t}.$$

- Functions

  void betval(char *t, double $val1$, double $val2$);

  void betdep(char *t, double $val1$, double $val2$, char *p);

  void betfun(char *t, double (*f)(), double (*g)());

  define a transition t with beta distribution with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, or the marking-dependent function f and g. The density is then, $\forall 0 < x < 1$,

$$\frac{1}{B(val1, val2)} x^{val1-1} (1-x)^{val2-1}.$$

- Functions

  void poisval(char *t, double $val1$, double $val2$);

  void poisdep(char *t, double $val1$, double $val2$, char *p);

  void poisfun(char *t, double (*f)(), double (*g)());

  define a transition t with Poisson distribution with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, or the marking-dependent function f and g. The probability to have value $k \times val2$, $\forall 0 \leq k$, is then given by $(val1)^k / k! e^{-val1 \times k}$.

- Functions

  void binoval(char *t, double $val1$, double $val2$,double $val3$);

  void binodep(char *t, double $val1$, double $val2$, double $val3$, char *p);

  void binofun(char *t, double (*f)(), double (*g)(), double (*h)());

define a transition t with binomial distributionwith respective parameters $val1$, $val2$ and $val3$, $val1$ times the number of tokens in place p, $val2$ times the number of tokens in place p and $val3$ times the number of tokens in place p, or the marking-dependent function f, g and h. The probability to have value $k \times val3$, $\forall 0 \leq k \leq val1$, is then given by

$$(val1, k)(1 - val2)^{val1-k} val2^k.$$

- Functions

  void negbval(char *t, double $val1$, double $val2$, double $val3$);

  void negbdep(char *t, double $val1$, double $val2$, double $val3$, char *p);

  void negbfun(char *t, double (*f)(), double (*g)(), double (*h)());

  define a transition t with negative binomial distribution with respective parameters $val1$, $val2$ and $val3$, $val1$ times the number of tokens in place p, $val2$ times the number of tokens in place p and $val3$ times the number of tokens in place p, or the marking-dependent function f, g and h. The probability to have value $k \times val3$, $\forall k \geq 0$, is then given by

$$\frac{\Gamma(val1 + k)}{\Gamma(val1)\Gamma(k + 1)}(1 - val2)^{val1} val2^k.$$

- Functions

  void hyperval(char *t, double $val1$, double $val2$, double $val3$);

  void hyperdep(char *t, double $val1$, double $val2$, double $val3$, char *p);

  void hyperfun(char *t, double (*f)(), double (*g)(), double (*h)());

  define a transition t with 2-stage hyperexponential distribution with respective rates $val1$, $val2$ and probability $val3$ to choose stage 1, $val1$ times the number of tokens in place p, $val2$ times the number of tokens in place p and $val3$ (as the probability to choose stage 1), or the marking-dependent function f, g and h.

- Functions

  void hypoval(char *t, double $val1$, double $val2$, double $val3$, double $val4$);

  void hypodep(char *t, double $val1$, double $val2$, double $val3$, double $val4$, char *p);

  void hypofun(char *t, double (*f)(), double (*g)(), double (*h)(), double (*i)());

define a transition t with 2 or 3-stages hypo-exponential distribution with number of stages $val1$ (2 or 3), and rates $val2$, $val3$ and $val4$, number of stages $val1$, and rates $val2$, $val3$ and $val4$ times the number of tokens in place p, or the marking-dependent functions f, g, h and i.

- Functions

  void erlval(char *t, double $val1$, double $val2$);

  void erldep(char *t, double $val1$, double $val2$, char *p);

  void erlfun(char *t, double (*f)(), double (*g)());

  define a transition t with a firing time following an Erlang distribution with respective parameters $val1$ and $val2$, $val1$ times the number of tokens in place p and $val2$ times the number of tokens in place p, and the marking-dependent function f and g. The first parameter denotes here the rate of the underlying exponential distribution and the second parameter the number of phases. Even if this last parameter is actually an integer, it must be given as a **double**.

- Functions

  void parval(char *t, double k, double $alpha$);

  void pardep(char *t, double k, double $alpha$, char *p);

  void parfun(char *t, double (*f)(), double (*g)());

  define a transition t with a firing time following a Pareto distribution with respective parameters k and $alpha$, k times the number of tokens in place p and $alpha$ times the number of tokens in place p, and the marking-dependent function f and g. Recall that the Pareto density f_{Par} is given by

  $$f_{Par}(x) = ak^a/(x+k)^{a+1}$$

  for $x \geq 0$ and $a, k > 0$.

- Functions

  void cauval(char *t, double $alpha$, double $beta$);

  void caudep(char *t, double $alpha$, double $beta$, char *p);

  void caufun(char *t, double (*f)(), double (*g)());

  define a transition t with a firing time following a Cauchy distribution with respective parameters $alpha$ and $beta$, $alpha$ times the number of tokens in place p and $betaa$ times the number of tokens in place p, and the marking-dependent function f and g. Recall that the Caucy density f_{Cau} is given by

  $$f_{Cau}(x) = \frac{\beta}{\pi(\beta^2 + (x-\alpha)^2)}$$

for $x > 0$, $\beta > 0$ and $-\infty < \alpha < \infty$.

4.3 Function: assert()

int assert(void);

A valid CSPL should contain function **assert()**. Function **assert** is a boolean marking-dependent function called by SPNP during the reachability graph construction to check the validity of each newly found marking. It returns either **RES_ERROR** if the marking is illegal or **RES_NOERR** if the marking is (thought to be) legal.

The check on the legality of a marking is performed using the same functions used to achieve marking dependency, namely **mark** and **enabled**. The check is by its own nature incomplete, since it is not usually feasible to specify all the conditions that must hold (or not hold) in a marking, but the more accurate the set of conditions is, the more confidence you have on the correspondence of the reachability graph with the real system.

For example, the following function **assert()**

```
assert() {
   if (mark("p_1") + mark("p_2") != N)
      return(RES_ERROR);
   return(RES_NOERR);
}
```

will stop the execution in a marking where the sum of the number of tokens in Places p_1 and p_2 is not N. If the execution is stopped, the program outputs information before exiting, which is useful in debugging the CSPL file. If the illegal marking is caused by an unforeseen sequence of transition firings, finding that sequence using the output information is usually a fast process even in large reachability graphs (of tens of thousands of markings).

However, this type of check is limited. It helps to detect the presence of illegal markings, or illegal firing sequences, but it can not detect the absence of legal markings, or legal firing sequences (which relates to the reachability set, or graph, as a whole, and cannot be checked while the reachability graph is being built). Practically, it is important to be able to perform checks of illegality as soon as possible, typically to debug a net which is supposed to be bounded, but it turns out to be not. The examination of the whole (infinite) reachability graph is out of the question since the program will terminate printing a message for insufficient memory.

4.4 Functions: ac_init() and ac_reach()

void ac_init(void);
void ac_reach(void);

A CSPL file must contain functions **ac_init()** and **ac_reach()**.

SPNP calls **ac_init()** before starting the reachability graph construction. The function

void pr_net_info(void);

is always called in function **ac_init()** to output information about the model under study to the "**.out**" file. This is especially useful when the number of places or transitions is defined at runtime. Calls to **bind** can also be used here to assign numeric values to parameters, as previously discussed in section, "Define and use parameters".

The function **ac_reach()** is instead called after the reachability graph construction has completed. The function

void pr_rg_info(void);

can be used in function **ac_reach()** to output information about the reachability graph to the "**.out**" file (Note: this does not affect the generation of the "**.rg**" file). Calls to **bind** can also be used here to assign numeric values to parameters as previously discussed. In addition, the function

void pr_parms(void);

can be called here to print the names of parameterized transitions along with the parameter names and current values to the "**.out**" file.

4.5 Function: ac_final()

void ac_final(void);

A CSPL file must contain function **ac_final()**. Function **ac_final()** is designed for a user to flexibly define outputs. CSPL provides a set of functions for this purpose (the simulation case will be treated separately in Chapter 6):

- **solve()**,
- **sens()**,
- **expected()**,
- **set_prob_init()**,
- **pr_mc_info()**,
- **pr_std_average()**,
- **pr_std_cum_average()**,
- **pr_expected()**,
- **pr_cum_expected()**,
- **pr_time_avg_expected()**,
- **pr_mtta()** and **pr_newmtta()**,
- **pr_mtta_fun()**,
- **pr_cum_abs()**,
- **pr_value()**,
- **pr_message()**.

Descriptions of these function are given below:

- **solve()**

  void solve(double t);

  Funciton **solve()** must be used **at least once** before any other function called to solve the Markov chain (numerically) at time t. The value given to *t* can be a positive real number for *transient analysis* or the value **INFINITY** for *steady state analysis*. Calls to **solve()** can be invoked repeatedly for different solution times and can be interleaved with user-requested output. Calls to **bind** can also be used here to (re)assign numeric values to parameters, as previously discussed; **solve()** must be called again to re-solve the new model[10].

[10] Whenever **solve** is called after a previous call to **bind**, the reachability graph and Markov chain will be reconstructed followed by another call to **ac_reach**.

- **void pr_mc_info()**

  void pr_mc_info(void);

  Function **void pr_mc_info()** can be called in **ac_final()** to output data about the Markov chain and its solution.

- **pr_std_average()**

  void pr_std_average (void);

  Function **pr_std_average** computes, for each place, the probability that it is not empty and its average number of tokens; for each timed transition, the probability that it is enabled and its average throughput. The average throughput $E[T_a]$ for Transition a is defined as

$$E[T_a] = \sum_{i \in R(a)} p(i) * \rho(a, i),$$

  where $R(a)$ is the subset of reachable markings that enable Transition a, $p(i)$ is the probability of marking i, and $\rho(a, i)$ is the rate of Transition a in marking i.

- **pr_expected()**

  void pr_expected(char *$string$, double (*$func$)());

  Function **pr_expected()** requires the specification of a string (which is written to the "**.out**" file) and of a marking-dependent reward function $func$ returning a double-precision floating point number.

  For example, the following function **ac_final()**

```
void ac_final() {
  ...
  pr_expected("utilization", util);
  ...
}
```

  will print

```
EXPECTED: utilization = 3.2
```

to the "**.out**" file if the expected value of the reward function **util()** is 3.2.

The **reward function util()** returning a **double** (double-precision floating point number), must have been defined prior to its usage in **pr_expected**, using the functions **mark()** and **enabled()** to express marking dependency. In addition to the functions **mark()** and **enabled()**, the function

double rate(char *t);

(as defined previously) can also be used to refer to the marking dependent rate of Transition t.

We provide an example to illustrate how to write a marking dependent reward functon. The following function **util()**

```
double util() {
  return(mark("p_1") * mark("p_7") + mark("p_3") * 1.2);
}
```

would define the utilization as the weighted (by the probability of each marking) average of the product of the number of tokens in Place p_1 and p_7 plus the number of tokens in Place p_3 times 1.2.

- **expected()**

  double expected(double (*$func$)());

  Function **expected** can be called in function **ac_final()**. The return value can be used in more complex expression (using **pr_expected** would print the value, but the value itself would not be made available and then be used in the function **ac_final**).

  Note: Apparently similar operations have different stochastic interpretations, hence different results, if performed at the event or at the expected value level. Continuing the previous example in function **pr_expected()**,

```
double ep_1() {
  return(mark("p_1"));
}

double ep_3() {
  return(mark("p_3"));
}

double ep_7() {
  return(mark("p_7"));
}
```

```
.......

void ac_final() {
  x = expected(ep_1) * expected(ep_7)
      + expected(ep_3) * 1.2;
  printf("%f",x);
}
```

will produce a different result from the one computed using **util()** because of the dependence existing (in general) between the number of tokens in p_1 and p_3.

- **pr_std_cum_average()**, **pr_cum_expected()** and **pr_time_avg_expected**

  Besides the expected values of the functions defined earlier, transient analysis also allows the computation of the expected accumulated values over the interval $[0, t)$ where t is the time point of interest. The corresponding functions are **pr_std_cum_average()** for computing the expected accumulated values and **pr_cum_expected()** for computing the expected accumulated value for user-defined functions. We can also compute the time-averaged expected values of functions using **pr_time_avg_expected** as:

  void pr_time_avg_expected (char *$string$, double (*$func$)());

  As an example, consider the following:

```
double  avail() {
  ...
}

void ac_final() {

   double time_point;

   solve(15.05);
   pr_expected("Inst. Availability",avail);
   pr_cum_expected("Total jobs lost",jobslost);

   for (time_point = 10.0;time_point <= 100.0;
        time_point += 10.0) {
      solve(time_point);
      pr_time_avg_expected("Interval Availability",avail);
   }

   pr_mtta("Mean time to failure");
}
```

Here, the instantaneous availability and total jobs lost are computed at time point $t = 15.05$. The for loop computes the instantaneous availability in the interval $[0, t)$, with t varying from 10 to 100 with an increment of 10.

- **pr_mtta()** and **pr_newmtta()**

  void pr_mtta(char *$string$); void pr_mtta(char *$string$);

  The function **pr_mtta** and **pr_newmtta()** computes the mean time to absorption for the SRN. The functions should be used only when the underlying CTMC has absorbing states. **pr_newmtta()** gives the same result but runs much faster.

- **pr_mtta_fun()**

  void pr_mtta_fun(char *$string$, double (*$func$)())

  computes the accumulated reward up to absorption using function $func$ to define which states are absorbing ($func = 0$) and the rewards of other states ($func! = 0$).

- **pr_cum_abs()**

  void pr_cum_abs(char *$string$, double (*$func$)());

  Function **pr_cum_abs()**, which is similar to **pr_mtta**, allows the user to compute the expected accumulated reward until absorption for a CTMC with absorbing states. To use this function, the corresponding reward rate should be specified.

- **set_prob_init()**

  void set_prob_init (double (*$func$)());

  Transient analysis of the underlying CTMC depends on the initial state probability vector. When the initial marking is tangible, the state corresponding to this marking has the initial probability of 1 and all the other states have an initial probability of 0. If the initial marking is vanishing, the initial probability vector over the states of the CTMC is automatically computed by the program. The user is also allowed to define the initial probability vector over the markings of the SRN using this function. At present, user-defined initial probability vector is allowed only over the set of tangible markings.

- **pr_value()**

  void pr_value(char *$string$, double $expr$**);**

Sometimes the user may desire to print values of functions that cannot be expressed as a simple reward definition, but as a function of the expected values of several reward functions. To facilitate this, SPNP provides a special function called **pr_value**($func$).

Here, $expr$ could be any expression which evaluates to a floating point number. For example, if we wish to compute the ratio of two expected values `expected(qlength)` and `expected(tput)` and print the result in the output file, we can specify the following:

```
pr_value("Expected Response Time",
     expected(qlength)/expected(tput));
```

This would print the following in the output file:

```
VALUE: Expected Response Time = 15.7
```

- **pr_message()**

  void pr_message(char *$string$);

  Function **pr_message** allows the user to print an arbitrary message in the .out file.

- **sens()**[11]

  void sens(char *$parameter$, ... (char *)0**);**

  If **IOP_SENSITIVITY** has the value **VAL_YES**, the sensitivities (derivatives) with respect to the model parameters will be provided for the measurements given by

  - **pr_std_average()**,
  - **pr_std_cum_average()**,
  - **pr_expected()**,
  - **expected()**,
  - **pr_time_avg_expected()**,
  - **pr_mtta()**,
  - **pr_cum_abs()**.

[11] See example CSPL file *sensi.c*.

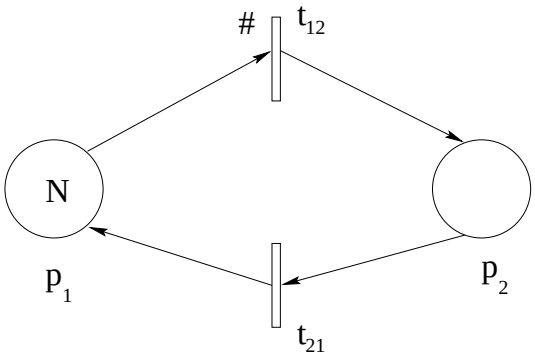


Figure 4.2: The SRN model

Currently, the reward functions can not be specified in terms of the defined parameters. Therefore, when computing the derivative of any measurement that requires a reward function, the reward accumulated in any state is assumed constant with respect to the parameters. By default, the sensitivities with respect to all defined parameters will be computed.

By making a call to **sens()** with a list of parameter names terminated with 0, one can define a set of enabled parameters that is a subset of all defined parameters. For example,

```
sens("alpha","beta","gamma",0);
```

will enable sensitivity analysis for only the parameters α, β, and γ, while

```
sens(0);
```

will produce an empty set of parameters thereby disabling sensitivity analysis, and

```
sens(ALL,0);
```

will enable sensitivity analysis for all defined parameters once again.

4.6 A complete example

As an example, consider an SRN (as shown in Figure 4.2) with two places, p_1 and p_2, and two transitions with exponentially distributed firing times, t_{12} and t_{21}. The rate of the first

transition is determined by the function $myval$ to be 7.3 times the number of tokens in Place p_1 while the rate of the second transition is the constant 1.0. The SPNP input file is shown as following:

```
# include "user.h"

/* Global variables */
int N;

void options() {
  iopt(IOP_SSMETHOD,VAL_SSSOR);
  iopt(IOP_PR_FULL_MARK,VAL_YES);
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC);
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_MC,VAL_YES);
  iopt(IOP_MC,VAL_CTMC);
  iopt(IOP_PR_PROB,VAL_YES);
  iopt(IOP_PR_RSET,VAL_YES);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  iopt(IOP_ITERATIONS,2000);
  iopt(IOP_CUMULATIVE,VAL_NO);
  fopt(FOP_ABS_RET_M0,0.0);
  fopt(FOP_PRECISION,0.00000001);
  N =  input("Please enter the token number N:");
}

void net() {

  place("p_1");
  place("p_2");

  trans("t_12");
  trans("t_21");

  init("p_1",N);

  /* timed transition */
  ratedep("t_12",7.3, "p_1");
  rateval("t_21",1.0);

  iarc("t_12","p_1");
  iarc("t_21","p_2");
  oarc("t_12","p_2");
  oarc("t_21","p_1");
}

int assert() {
  return RES_NOERR;
}
```

```
void ac_init() {
}

void ac_reach() {
}

/*reward rate */
double tokenNo() {
  return(mark("p_1"));
}

double rd() {
  return ((mark("p_2")==0) ? 1:0);
}


void ac_final() {
  solve(INFINITY);
  pr_expected("Average Token Number in place p_1:", tokenNo);
  pr_expected("steady state prob. that place p_2 is empty: ", rd);
}
```

Chapter 5

Specialized Output Functions

A state of a Markov chain is called *transient* if, during an infinitely long observation time period, the system visits this state only finitely often. In other words, there is a nonzero probability that the system will never return to this state. A state is defined as *recurrent* if the system visits this state infinitely often during an infinitely long observation period. A recurrent state is called *positive recurrent* or *non-null recurrent* if its average recurrence time is finite. Otherwise, it is defined as a *null recurrent* state.

A Markov chain is *irreducible* if every state can be reached by every other state. A CTMC is *ergodic* if it is irreducible and positive recurrent.

SPNP was initially aimed at the steady-state solution of SRNs whose underlying CTMC is ergodic. There are a number of measures which could be considered unusual, but closely related to steady-state. In particular, they do not require the implementation of a new solver; they can be computed either from the steady-state probabilities, or by solving a slightly different (non-homogeneous) linear system.

These measures were defined and implemented to perform decomposition–iteration techniques allowing the approximate solution of SRNs whose state-space is too large to be studied directly [9, 10]. They are **accumulated**, **pr_accumulated**, **hold_cond**, **pr_hold_cond**, and **set_prob0**.

double accumulated($mfunction$**,**$function$**)**
double (*$mfunction$**)();**
double (*$function$**)();**

void pr_accumulated($string$**,**$function$**)**
char *$string$**;**
double (*$function$**)();**

respectively return and print the expected value of the "accumulated reward up to absorption", according to some initial state probability distribution, reward rate assignment, and absorbing marking definition. In the **accumulated** function, the reward rate assignment is specified by $function$ and the absorbing markings are defined by $mfunction$; if $mfunction$ evaluates to 0.0 then the marking is considered absorbing. In the **pr_accumulated** function, $function$ specifies *both* the reward and the absorbing marking; if $function$ evaluates to 0.0 then the marking is considered absorbing. The specification of the absorbing markings requires some attention. Since the SRNs normally managed by the package are

ergodic, no absorbing markings may be present. The underlying stochastic process is then modified (for the computation of this measure only) so that markings whose reward is null are assumed absorbing (their outgoing arcs in the Markov chain are ignored). If absorbing markings do indeed exist in the original SRN, $function$ (as well as $mfunction$) must evaluate to zero in them (otherwise the accumulated reward would be infinite). Each call to **accumulated** (or **pr_accumulated**) requires the solution of a non-homogeneous linear system having as many variables as the non-zero-reward markings, so it can be expensive.

The results obtained from calls to **accumulated** or **pr_accumulated** is dependent on the initial state probability vector. When the initial marking is tangible, the state corresponding to this marking has the initial probability of 1 and all the other states have an initial probability of 0. If the initial marking is vanishing, the initial probability vector over the states of the CTMC is automatically computed by the program. Alternatively, the user is allowed to define the initial probability vector over the markings of the SRN using the function

void set_prob0($scale$**,**$function$**)**
int $scale$**;**
double (* $function$**)();**

where $scale$ is either **VAL_YES** or **VAL_NO** and the reward rate assignment is given by $function$. At present, user-defined initial probability vector is allowed only over the set of tangible markings. Unless the default value for the initial probability is desired, **set_prob0** must be called before each call to **accumulated** or **pr_accumulated**. Let's define ρ_i as the value returned by $function$ on marking i and π_i as the steady-state probability for marking i ($\pi_i = 0$ if marking i is vanishing). A call to **set_prob0** with $scale$ equal to **VAL_NO** defines the initial state probability for state i to be proportional to ρ_i:

$$\pi(0)_i = \frac{\rho_i}{\sum_j \rho_j}.$$

A call to **set_prob0** with $scale$ equal to **VAL_YES** defines the initial state probability for state i to be proportional to $\rho_i \pi_i$:

$$\pi(0)_i = \frac{\rho_i \pi_i}{\sum_j \rho_j \pi_j}.$$

The definition of this second function may at first seem arbitrary; it is instead both useful and intuitive. Assume that, given an ergodic SRN in steady-state, we want to know how long we need to wait before a token arrives in Place p. The following portion of CSPL accomplishes this:

```
double one()    { return(1.0); }
```

```
double empty() { return( mark("p") > 0 ? 0.0 : 1.0 ); }
double full()  { return( mark("p") > 0 ? 1.0 : 0.0 ); }
.......
void    ac_final() {
     set_prob0(VAL_YES,empty);
     pr_accumulated("Wait time",full);
     set_prob0(VAL_YES,one);
     pr_accumulated("Wait time",full);
}
```

The first output gives the waiting time given that no token is in p, while the second output gives the unconditional waiting time (that is, including the possibility that a zero waiting time is required, when a token is already in p).

If **IOP_SENSITIVITY** has the value **VAL_YES**, the sensitivities (derivatives) with respect to the model parameters will be provided for the measurements given by **accumulated** and **pr_accumulated**. See the previous section for the definitions of **parm**, **useparm**, and **bind**, which are required for sensitivity analysis.

Functions **hold_cond** and **pr_hold_cond** respectively compute and print the expected time a condition holds true or false in steady-state:

void hold_cond($cond$**,**$times$**)**
int (*$cond$**)();**
double $times[2]$

void pr_hold_cond($string$**,**$cond$**)**
char *$string$**;**
int (*$cond$**)();**

$cond$ must be a marking-dependent function returning **VAL_YES** if the condition holds in the marking, **VAL_NO** otherwise. On return, $times[\mathbf{VAL_YES}]$ and $times[\mathbf{VAL_NO}]$ respectively contain the expected length of time the condition holds or does not hold in steady-state. The idea behind this measure is to be able to condense a large Markov chain into a two-state process. Normally the process is not a Markov chain, but the two-state Markov chain whose transition rates are $1/times[\mathbf{VAL_YES}]$ and $1/times[\mathbf{VAL_NO}]$ can at least be considered an approximate representation of it. The description of how this measure is computed gives additional insight. Define S_y and S_n to be the sets of markings where the condition is true and false respectively and define T to be the set of tangible markings. If **IOP_MC** has value **VAL_CTMC**, $times[\mathbf{VAL_YES}]$ and $times[\mathbf{VAL_NO}]$

are computed respectively as

$$\frac{\sum_{k \in S_y \cap T} \pi_k}{\sum_{i \in S_y \cap T, j \in S_n \cap T} \pi_i \lambda_{i,j}} \quad \text{and} \quad \frac{\sum_{k \in S_n \cap T} \pi_k}{\sum_{i \in S_n \cap T, j \in S_y \cap T} \pi_i \lambda_{i,j}}$$

where $\lambda_{i,j}$ is the transition rate from marking i to marking j. If **IOP_MC** has value **VAL_DTMC**, $times[\mathbf{VAL_YES}]$ and $times[\mathbf{VAL_NO}]$ are computed respectively as

$$\frac{\left(\sum_{k \in S_y \cap T} \pi_k\right)\left(\sum_{k \in T} p_k h_k\right)}{\sum_{i \in S_y, j \in S_n} p_i \alpha_{i,j}} \quad \text{and} \quad \frac{\left(\sum_{k \in S_n \cap T} \pi_k\right)\left(\sum_{k \in T} p_k h_k\right)}{\sum_{i \in S_n, j \in S_y} p_i \alpha_{i,j}}$$

where $\alpha_{i,j}$ is the transition probability from marking i to marking j, p_j is the steady-state probability of marking j for the DTMC, and h_k is the holding time in state k for the CTMC.

It is interesting to notice that DTMC and CTMC solution may give different results for this measure. The reason is not due to an error nor to numerical roundoff or truncation. Rather, it is intrinsic to the different approaches. If the condition holds in tangible markings m_1 and m_3 and it does not hold in vanishing marking m_2, a path (m_1, m_2, m_3) in the reachability graph is treated differently by the two approaches. The DTMC solution considers the holding time as terminated and restarted every time the path is traversed, while the CTMC solution does not know that the condition stops holding, even if for a null amount of time, when a transition from m_1 to m_3 occurs (this information is discarded together with m_2 when eliminating the vanishing markings). The holding time computed by the DTMC solution can be shorter than the one computed by the CTMC solution. In practically all interesting applications, the condition holds or does not hold for a positive amount of time with probability one, so no inconsistencies can arise.

Chapter 6

Discrete-Event Simulation

In addition to a numeric-analytic solution, SPNP allows to use discrete-event simulation to study the behavior of a system at or up to a point of time. The specification of the SPN model is done in CSPL. For simulation, however, the firing time distributions can be different from exponential distribution. Generally, three functions are available to define one distribution with different kinds of parametrizations: constant (with `xxxval()`), marking dependent (with `xxxdep()`), and function dependent (with `xxxfun()`). The list of distributions and their definition functions are provided below (For more details, see Section 4.2.10):

- Exponential: `rateval(), ratedep(), ratefun()`.
- Constant (Or deterministic, including zero, that is, immediate transitions): `detval(), detdep(), detfun()`.
- Uniform: `unifval(), unifdep(), uniffun()`.
- Geometric: `geomval(), geomdep(), geomfun()`.
- Weibull: `weibval(), weibdep(), weibfun()`.
- (truncated) Normal: `normval(), normdep(), normfun()`.
- Lognormal: `lognval(), logndep(), lognfun()`.
- Erlang: `erlval(), erldep(), erlfun()`.
- Gamma: `gamval(), gamdep(), gamfun()`.
- Beta: `betval(), betdep(), betfun()`.
- (truncated) Cauchy: `cauval(), caudep(), caufun()`.
- Binomial: `binoval(), binodep(), binofun()`.
- Poisson: `poisval(), poisdep(), poisfun()`.
- Pareto: `parval(), pardep(), parfun()`.
- Hyperexponential (2-stage): `hyperval(), hyperdep(), hyperfun()`.

- Hypoexponential (2 or 3-stage): `hypoval(), hypodep(), hypofun()`.

The following types will be added to the package later:

- Negative Binominal: `negbval(), negbdep(), negbfun()`.
- Cox2
- Loglogistic
- Defective exponential
- Triangular

When using simulation, the reachability graph is not generated, but the **assert** specification can still be given: it is checked in each marking encountered during the simulation.

The output of SPN in this case is analogous to the one obtained from an analytic-numerical solution, but with confidence intervals instead of point values. For FSPNs, simulation is the only solution method implemented in SPNP.

6.1 Standard discrete event simulation

The following information is specified in the CSPL file, normally using calls to $iopt()$ or $fopt()$:

- **Simulation specification**
  IOP_SIMULATION specifies if the simulation procedure will be used. Default value is **VAL_NO**.
- **Discrete event simulation method**
  IOP_SIM_RUNMETHOD specifies the simulation method. Value
  VAL_REPL is specified if independent replications are used and value **VAL_BATCH**) if it is batches [31]. Other values, for other simulation methods, are possible and will be explained in next sections.
- **Simulation length**
  FOP_SIM_LENGTH is the length of each simulation run, in simulated time, to be specified with a call to $fopt()$ (in the case where batches are used, it represents the length of each batch). If no value is specified, it is possible to use calls to **at_time** or **cum_time** in function **ac_final** instead.

- **Mode of data collection**
  IOP_SIM_STD_REPORT specifies that the results will be displayed in the .out file and the call of **pr_message(char *msg)** in **ac_final()** allows to print a message in the .out file. . **IOP_SIM_CUMULATIVE** allows the data to be collected cumulatively (from zero to **FOP_SIM_LENGTH**). The simulator collects four standard measures:

  – Probability that a place is not empty.
  – Average number of tokens in a place.
  – Probability that a transition is enabled.
  – Throughput of a transition.

  The default value of **IOP_SIM_CUMULATIVE** is **VAL_YES**.

  Another way to collect statistics is by calling in **ac_final()** functions

  – **pr_expected(char *msg, double (*f)())**, which computes the average instantanenous value at time **IOP_SIM_LENGTH** for the user defined function f;
  – **pr_cum_expected(char *msg, double (*f)())** which computes the average cumulative value for the user defined function f from time 0 to time given by **IOP_SIM_LENGTH**.

  Thus option **IOP_SIM_CUMULATIVE** saves the call of a lot of functions **pr_cum_expected()** but does not prevent the user from computing instantaneous measures (function **ac_final()** can be empty if the values the user is looking for are an output of **IOP_SIM_CUMULATIVE**).

- **Data confidence interval**
  FOP_SIM_CONFIDENCE specifies the confidence to be used when computing the confidence intervals. Possible values are $90\%, 95\%, 99\%$, and the default value is 95%.

- **Number of iterations**
  IOP_SIM_RUNS specifies the maximum number of simulation runs to be performed, to obtain meaningful statistics. **FOP_SIM_ERROR** specifies the target half-width of the confidence interval, relative to the point estimate. You can provide the exact number of runs (or batches). The default value is 0. If you wish to let SPNP perform as many runs as needed to achieve the specified relative error (which should then be strictly between zero and one), you should assign **IOP_SIM_RUNS** to 0 (or not assign anything to it) and then specify **FOP_SIM_ERROR**. If neither is defined, the default is then to run until a 10% relative error is reached (at least five runs are always performed in this case).

- **Random generator**
  IOP_SIM_SEED allows to change the seed of the random generator.

6.1.1 Current limitations of the simulator

The current limitations of the simulator are the following:

- The FSPNs iplemented are linear FSPNS, which means that the rates of the flows from or in fluid places are linear between two firing of transitions or until a bound has been hit in a fluid place.
- The guards, the flows can depend only on the discrete marking, the bounds of fluid places and thresholds in fluid places (and not on the whole state space) and function **fbreak()** doesn't allow yet to take into account the increasing or decreasing property of the fluid level to modify consequently the rates or guards.
- The cumulative measures to be computed can not involve a fluid place as they are computed by the sum of the measure in the current state multiplied by the time to the next event (a firing of a transition or a fluid event, i.e, a bound is hit in a fluid place).

6.1.2 Examples

See Section 10.13 and Section 10.14.

6.2 Importance splitting for rare events

The discrete event simulation described above is a very powerful tool but is inefficient to examine rare events. To do so, we can use importance splitting techniques [34], which means defining thresholds and splitting the simulation path when the thresholds are hit. The aim of these methods in this software is to compute $P(\#p \geq x \ in \ [0,T])$, the probability to have more than a given amount a tokens (or fluid) in a place in the time interval $[0,T]$, or $P(\tau_{p,x} < \min(T,\tau_0))$, the probability that $\tau_{p,x}$, the first time that $\#p \geq x$, is less than the simulation run T and less than τ_0, the return time to the initial state.

An advantage with respect to the discrete event simulation of the previous section is that here the measures can be computed on fluid places.

The usual simulation options **IOP_SIM_RUNS** and **FOP_SIM_LENGTH** are still used by the importance splitting estimation.

The use of importance splitting techniques is specified by the option **IOP_SIM_RUNMETHOD**. Its is set to **VAL_RESTART** if we use RESTART, estimating then $P(\#p \geq x \ in \ [0,T])$, and to **VAL_SPLIT** if we use splitting, estimating then $P(\tau_{F,x} < \min(T,\tau_0))$. For each

method (RESTART or splitting), the importance splitting procedure is called in **ac_final** by inserting **splitting(***name_of_place* p**,**x**)**.

Two specific options exist, one for splitting, **IOP_SPLIT_LEVEL_DOWN**, which determines the number d of levels the path is stopped if it goes d levels down, and one for RESTART, **IOP_SPLIT_RESTART_FINISH**, which means that each retrial is finished at simulation time T (see [34]).

To determine the thresholds where the path will be split, there are two possibilities:

- either they are determined by the user, by assigning in **options** the option **IOP_SPLIT_PRESIM** to **VAL_NO**, then assigning the number of thresholds to **IOP_SPLIT_NUMBER** and finally setting the thresholds values in table **FOP_SPLIT_THRESHOLDS**. If this table is not specified, the thresholds are chosen uniformly between the initial value in place p and value x.
- Or he runs a presimulation (**IOP_SPLIT_PRESIM=VAL_YES**). We run then at each level a standard discrete event simulation, using number of independent runs is given by **IOP_SPLIT_PRESIM_RUNS**.

For further details about importance splitting and its implementation in SPNP, see [34].

6.2.1 Example

See Section 10.15.

6.3 Importance sampling

Importance sampling is a method particularly efficient to estimate rare events, though it can be used to improve the accuracy of every simulation. The basic idea is to modify the firing time distribution in order to reduce the variance of the measure we wish to compute. The bias induced by this change of measure is then corrected by the introduction of a function called the likelihood ratio (tranparent in the outputs).

Importance sampling is called by setting **IOP_SIM_RUNMETHOD** to **VAL_IS**. The net is defined as usual with the corresponding firing distributions. In the current implementation, the only distributions for which the sampling distribution can be changed (and the list of new distributions which can be used) is the following:

- Exponential.

- Uniform.
- Weibull.
- Erlang.
- (truncated) Cauchy.
- Pareto.
- Hyperexponential (2-stage).
- Probabilities.

This is done by calling the following functions (after the transition has been defined)

void rateval_is(char *t, double val);
void ratedep_is(char *t, double *val*, char *p);
void ratefun_is(char *t, double (*$func$());
void unifval_is(char *t, double $lower$, double $upper$);
void unifdep_is(char *t, double $lower$, double $upper$, char *p);
void uniffun_is(char *t, double (*f)(), double (*g)());
void weibval_is(char *t, double $val1$, double $val2$);
void weibdep_is(char *t, double $val1$, double $val2$, char *p);
void weibfun_is(char *t, double (*f)(), double (*g)());
void hyperval_is(char *t, double $val1$, double $val2$, double $val3$);
void hyperdep_is(char *t, double $val1$, double $val2$, double $val3$, char *p);
void hyperfun_is(char *t, double (*f)(), double (*g)(), double (*h)());
void erlval_is(char *t, double $val1$, double $val2$);
void erldep_is(char *t, double $val1$, double $val2$, char *p);
void erlfun_is(char *t, double (*f)(), double (*g)());
void parval_is(char *t, double k, double $alpha$);
void pardep_is(char *t, double k, double $alpha$, char *p);
void parfun_is(char *t, double (*f)(), double (*g)());
void cauval_is(char *t, double $alpha$, double $beta$);
void caudep_is(char *t, double $alpha$, double $beta$, char *p);

void caufun_is(char *t, double (*f)(), double (*g)());

void probval_is(char *t, double val**);**

void probdep_is(char *t, double *val*, **char *p);**

void probfun_is(char *t, double (*$func$());

The parameters are defined in the same way than when defining the actual distributions of the transitions.

It is sometime interesting to modify the parameters of the importance sampling distribution if some conditions are verified (for example if a set of states is reached). Such conditions are given by the user by calling (many times if there are many conditions) function

resampling(f**)**.

. Then the firing times are resampled using the new paprameters. This is called dynamic importance sampling.

6.4 Regenerative simulation

Regenerative simulation is used to estimate steady-state measures. It is called by setting **IOP_SIM_RUNMETHOD** to **VAL_REG**. In the current implementation, the user must specify the regenerative state as the initial state and be sure that this state is regenerative. The number of used regenerative cycles may be specified by **IOP_SIM_RUNS** or the desired precision by **FOP_SIM_ERROR**. **FOP_SIM_LENGTH** is not used for this type of simulation.

6.5 Regenerative simulation with importance sampling

Regenerative simulation with importance sampling is also used to estimate steady-state measures. It is called by setting **IOP_SIM_RUNMETHOD** to **VAL_ISREG**. It combines regenerative simulation with importance sampling to speed up the simulation.

Chapter 7

Available Options

Tables 7.1, 7.2, 7.3 and 7.4 lists all options and their legal and default values in CSPL. Their usage will be given in the following sections.

Name	Values	Default
IOP_PR_RSET	VAL_YES, VAL_NO, VAL_TAN	VAL_NO
IOP_PR_RGRAPH	VAL_YES, VAL_NO	VAL_NO
IOP_PR_MARK_ORDER	VAL_CANONIC VAL_LEXICAL VAL_MATRIX	VAL_CANONIC
IOP_PR_MERG_MARK	VAL_YES, VAL_NO	VAL_YES
IOP_PR_FULL_MARK	VAL_YES, VAL_NO	VAL_NO
IOP_USENAME	VAL_YES, VAL_NO	VAL_NO
IOP_PR_MC	VAL_YES, VAL_NO	VAL_NO
IOP_PR_DERMC	VAL_YES, VAL_NO	VAL_NO
IOP_PR_MC_ORDER	VAL_FROMTO, VAL_TOFROM	VAL_FROMTO
IOP_PR_PROB	VAL_YES, VAL_NO	VAL_NO
IOP_PR_PROBDTMC	VAL_YES, VAL_NO	VAL_NO
IOP_PR_DOT	VAL_YES, VAL_NO	VAL_NO

Table 7.1: Available options for intermediate files

Name	Values	Default
IOP_MC	VAL_CTMC, VAL_DTMC	VAL_CTMC
IOP_SSMETHOD	VAL_SSSOR, VAL_GASEI, VAL_POWER	VAL_SSSOR
IOP_SSDETECT	VAL_YES, VAL_NO	VAL_YES
FOP_SSPRES	non-negative **double**	0.25
IOP_TSMETHOD	VAL_TSUNIF, VAL_FOXUNIF	VAL_FOXUNIF
IOP_CUMULATIVE	VAL_YES, VAL_NO	VAL_YES
IOP_SENSITIVITY	VAL_YES, VAL_NO	VAL_NO
IOP_ITERATIONS	non-negative **int**	2000
FOP_PRECISION	non-negative **double**	0.000001

Table 7.2: Available analytic-numeric solution options

Name	Values
IOP_SIMULATION	VAL_YES, VAL_NO
IOP_SIM_RUNS	non-negative **int**
IOP_SIM_RUNMETHOD	VAL_REPL, VAL_BATCH, VAL_RESTART VAL_SPLIT, VAL_IS, VA
IOP_SIM_SEED	non-negative **int**
IOP_SIM_CUMULATIVE	VAL_YES, VAL_NO
IOP_SIM_STD_REPORT	VAL_YES, VAL_NO
IOP_SPLIT_LEVEL_DOWN	non-negative **double**
IOP_SPLIT_PRESIM	VAL_YES, VAL_NO
IOP_SPLIT_NUMBER	non-negative **double**
IOP_SPLIT_RESTART_FINISH	VAL_YES, VAL_NO
IOP_SPLIT_PRESIM_RUNS	non-negative **double**
FOP_SIM_LENGTH	non-negative **double**
FOP_SIM_CONFIDENCE	non-negative **double**
FOP_SIM_ERROR	non-negative **double**

Table 7.3: Available simulative options

Name	Values	Default
IOP_ELIMINATION	VAL_REDONTHEFLY, VAL_REDAFTERRG, VAL_REDNEVER	VAL_REDONTHEFLY
IOP_OK_ABSMARK	VAL_YES, VAL_NO	VAL_NO
IOP_OK_VANLOOP	VAL_YES, VAL_NO	VAL_NO
IOP_OK_TRANS_M0	VAL_YES, VAL_NO	VAL_YES
IOP_OK_VAN_M0	VAL_YES, VAL_NO	VAL_YES
FOP_ABS_RET_M0	non-negative **double**	0.0
IOP_DEBUG	VAL_YES, VAL_NO	VAL_NO
FOP_FLUID_EPSILON	non-negative **double**	0.000001
FOP_TIME_EPSILON	non-negative **double**	0.000001

Table 7.4: Miscellaneous options

7.1 Options for intermediate files

IOP_PR_RSET and IOP_PR_RGRAPH

These options specify whether the reachability set and graph should be printed. In addition to **VAL_YES** and **VAL_NO**, **VAL_TAN** can be used for **IOP_PR_RSET**, which indicates that only the tangible markings should be printed.

IOP_PR_MARK_ORDER

This option specifies the order in which the markings are printed.

- With **VAL_CANONIC** order, markings are printed in the order they are found, in a breadth-first search starting from the initial marking, and in increasing order of enabled transitions indices. It is the most natural order and it is particularly helpful when debugging the SRN.
- With **VAL_LEXICAL** order, markings are printed in increasing order, where markings are compared as words in a dictionary, for example (2_T 3:2 4:1 5:1) comes before (3_A 3:2 4:3 6:1). This order may be useful when searching for a particular marking in a large "**.rg**" file. With the **VAL_CANONIC** order, an editor with search capabilities is usually adequate for this purpose.
- With **VAL_MATRIX** order, markings are printed in the same order as the states of the two internal Markov chains: the DTMC corresponds to the vanishing markings, and the CTMC corresponds to the tangible markings. This corresponds to the following ordering: vanishing, tangible non-absorbing, and tangible absorbing, each of these groups ordered in canonical order.

IOP_PR_MERG_MARK

This option specifies whether the tangible and vanishing markings should be printed out in one merging list or two separate lists.

IOP_PR_FULL_MARK

This option specifies whether the markings are printed in long format, where some of the markings have zero number of tokens in all the places; or short format, where for each printed out marking, there is at least one place which has non-zero tokens.

IOP_USENAME

This option specifies whether the names should be used to indicate the places and transitions involved when printing the reachability set and graph, instead of the index (a small integer starting at 0). Using names generates a large "**.rg**" file but it is useful when debugging a SRN.

IOP_PR_MC

This option specifies whether the "**.mc**" file should be generated or not.

IOP_PR_DERMC

This option specifies whether the derivative with respect to each defined $parmname$ files should be generated or not in the file "**.mc**"

IOP_PR_MC_ORDER

This option specifies whether the transition matrix (if **VAL_FROMTO**) or its transpose (if **VAL_TOFROM**) should be printed in the "**.mc**" file.

IOP_PR_PROB

This option specifies whether the "**.prb**" file is generated or not.

IOP_PR_PROBDTMC

This option specifies whether the "**.prbdtmc**" file is generated or not.

IOP_PR_DOT

This option specifies whether the "**.dot**" file is generated or not.

7.2 Options for analytic-numeric solution

IOP_MC

This option specifies the solution approach.

- Using **VAL_CTMC** will transform the SRN into a CTMC.
- Using **VAL_DTMC** will use an alternative solution approach, where the vanishing marking are not eliminated and an embedded DTMC is solved instead. In this case, the first index in the "**.mc**" file is $-n$, if there are n vanishing markings, not 0.

SPNP can perform transient and sensitivity analysis only by reducing the SRN to a CTMC. Hence this option should be set to **VAL_CTMC** when these types of analysis are needed.

IOP_SSMETHOD

This option specifies the (numerical) steady-state solution method for the process:

- **VAL_SSSOR** for Steady-State SOR (Successive Overrelaxation)[29].
- **VAL_GASEI** for Steady-State Gauss-Seidel[29].
- **VAL_POWER** for Steady-State Power-Series Algorithm [35]

SOR is usually the fastest method, but there are cases where SOR does not converge, while Gauss-Seidel converges, and vice versa. The Power-Series Algorithm has a better convergence performance than the other two, but the the rate is much slower.

IOP_SSDETECT

This option specifies whether or not we wish to use steady-state detection in transient analysis for stiff Markov chains. In this option, the numerical transient probabilities of underlying stiff CTMC are obtained through the uniformization with steady-state detection of the underlying DTMC and computation of Poission probabilities using the method of Fox and Glynn. The readers can find details in [18].

- **VAL_YES** for transient solution using steady-state detection (default)
- **VAL_NO** for not using steady-state detection

FOP_SSPRES

This option specifies the required precision for the steady state detection.

IOP_TSMETHOD

This option specifies the (numerical) transient-state solution method for the CTMC:

- **VAL_TSUNIF** for Transient Solution using Standard Uniformization
- **VAL_FOXUNIF** for Uniformization using the Fox and Glynn method for computing the Poisson probabilities.

IOP_CUMULATIVE

This option specifies whether cumulative probabilities should be computed.

IOP_SENSITIVITY

This option specifies whether sensitivity analysis should be performed.

- If **VAL_YES** is specified, **IOP_MC** must have value **VAL_CTMC** and **IOP_ELIMINATION** must have value **VAL_REDONTHEFLY**.

IOP_ITERATIONS

This option specifies the maximum number of iterations allowed for the numerical solution. Any nonnegative integer can be specified.

FOP_PRECISION

This option specifies the minimum precision required from the numerical solution. The numerical solution will stop either if the precision is reached, or if the maximum number of iteration is reached. Both the reached precision and the actual number of iterations are always output in the "**.prb**" file, so you can (and should) check how well the numerical algorithm performed.

7.3 Options for simulative solution

IOP_SIMULATION

This option indicates whether the system is solved numerically or simulated.

IOP_SIM_RUNS

This option specifies the number of simulator runs (or batches). It can either be a positive integer or zero. In the latter case, SPNP runs until **FOP_SIM_ERROR** is reached.

IOP_SIM_RUNMETHOD

This option specifies the simulation method which will be used:

- **VAL_REPL** if we are using the standard discrete vent simulation with independent replications
- **VAL_BATCH** if we are using the standard discrete vent simulation with batches (and then a single run).
- **VAL_RESTART** if we are using Restart.
- **VAL_SPLIT** if we are using splitting.
- **VAL_IS** if we are using importance sampling. (In implementation).
- **VAL_REG** if we are using regenerative simulation. (In implementation).
- **VAL_ISREG** if we are using regenerative simulation with importance sampling. (In implementation).
- **VAL_THIN** if we are using thinning with independent replications. (In implementation).
- **VAL_BATHIN** if we are using thinning with batches (and then a single run). (In implementation).
- **VAL_ISTHIN** if we are using thinning with importance sampling. (In implementation).

The length of the replications or of the batches is given by **FOP_SIM_LENGTH** (for the regenerative simulation,**FOP_SIM_LENGTH** is not used) .

his option specifies, when **IOP_SIM_SPLIT**=**VAL_YES**, if we are using the RESTART method or the splitting one (which don't estimate the same thing).

IOP_SIM_SEED

This option specifies the value of the seed of the random generator. Default value is 52836.

IOP_SIM_CUMULATIVE

This option specifies whether to collect data as time averages or point-of-time estimates.

IOP_SIM_STD_REPORT

This option specifies whether the standard report is printed to the .out file in simulation.

IOP_SPLIT_LEVEL_DOWN

This option is specific to the splitting method (**IOP_SIM_SPLIT_RESTART**=**VAL_NO**). It specifies the number d of levels the simulation must cross down to stop the simulation path (to reduce the computational time).

IOP_SPLIT_RESTART_FINISH

This option is specific to the RESTART method (i.e., **IOP_SIM_SPLIT_RESTART** =**VAL_NO**). It specifies that every path will continue up to the simulation time given by **FOP_SIM_LENGTH** or up to reaching an upper thershold (not only the last one as in the usual RESTART method).

IOP_SPLIT_PRESIM

This option specifies, when we are using importance splitting techniques, if we determine the thresholds by running a presimulation.

IOP_SPLIT_PRESIM_RUNS

The option specifies the number of independent paths to use to estimate each threshold, when the presimulation is required for importance splitting methods.

IOP_SPLIT_NUMBER

This option specifies, if IOP_SPLIT_PRESIM=**VAL_NO**, the number of thresholds must be used to apply importance splitting methods. These thresholds must be specified in table **FOP_SPLIT_THRESHOLDS[1..IOP_SPLIT_NUMBER]** in function **setup()**.

FOP_SIM_LENGTH

This option specifies the time to run for each simulation iteration. No default is assumed, so the user has to specify a value for it whenever **IOP_SIMULATION** is **VAL_YES**.

FOP_SIM_CONFIDENCE

This option specifies the required confidence for the simulation, a number between zero and one. Currently, only 90%, 95%, or 99% can be specified.

FOP_SIM_ERROR

If **IOP_SIM_RUNS** is 0 or left unspecified, the simulator takes this error as the stopping criterion, it will run until the error precision is reached. It must be a value between 0.0 and 1.0.

7.4 Miscellaneous options

IOP_ELIMINATION

This option specifies the method by how vanishing markings are managed and eventually eliminated.

- Specifying **VAL_REDNEVER** means that the stochastic process being considered explicitly regards the vanishing markings as ordinary states. With this option, only a steady-state solution is possible. Using an embedded DTMC, measures related to immediate transitions are computed, and vanishing (non-absorbing) loops present no problems.
- Specifying **VAL_REDAFTERRG** means that the reachability graph constructed includes explicitly the vanishing markings, but these are then eliminated numerically before generating the underlying CTMC. With this option, any type of solution is possible, and vanishing (non-absorbing) loops present no problems, but measures related to immediate transitions are not computed.
- Finally, specifying **VAL_REDONTHEFLY** means that vanishing markings are eliminated during the reachability graph construction. With this option, any type of solution is possible, but vanishing (non-absorbing) loops are considered an error and measures related to immediate transitions are not computed.

Users are strongly encouraged to use **VAL_REDONTHEFLY**, as this usually results in the fastest solution and the lowest memory requirements. However, there are rare pathological cases where this option will actually results in larger memory requirements than **VAL_REDNEVER** [7].

IOP_OK_ABSMARK, IOP_OK_VANLOOP, IOP_OK_TRANS_M0, and IOP_OK_VAN_M0

These options specify respectively whether absorbing markings, transient vanishing loops, a transient initial marking, and a vanishing initial marking are acceptable or not.

- If **VAL_NO** is specified, the program will stop if the condition is encountered.
- If **VAL_YES** is specified, the program will signal such occurrences, but it will continue the execution, if it is possible.

FOP_ABS_RET_M0

This option specifies the value of the rate from each absorbing marking back to the initial marking. If this rate is positive, these markings will not correspond to absorbing states in the CTMC. This is useful to model a situation that would otherwise require a large number of transitions to model this "restart". Of course, the numerical results will depend on the value specified for this option.

IOP_DEBUG

Setting this option to **VAL_YES** causes SPNP to output (on the "**stderr**" stream) the markings as they are generated, and the transitions that enabled them. This feature is extremely useful when debugging an SRN.

FOP_FLUID_EPSILON

This option specifies the ε for which to values (concerning the level in a fluid place) are considered identical if they difference is smaller than ε. This value is introduced to prevent numerical round-off mistakes.

FOP_TIME_EPSILON

This option specifies the ε for which to values of time are considered identical if they difference is smaller than ε. This value is introduced to prevent numerical round-off mistakes and is needed when using importance splitting simulation methods.

Chapter 8

Format of the Intermediate Files

This section explains how to interpret the data in the intermediate files generated during the analysis of an SRN.

8.1 The ".rg" file

This file describes the reachability graph corresponding to the SRN. It can be generated if the options **IOP_PR_RSET** and **IOP_PR_RGRAPH** are set to **VAL_YES**. The format of the information is as the following:

```
_nplace = <number of places>;
_ntrans = <number of transitions>;
_places =
        <pl>: <place name>;
        ..................
        <pl>: <place name>;
_transitions =
        <tr>: <transition name>;
        .......................
        <tr>: <transition name>;
_ntanmark = <number of tangible non-absorbing markings>;
_nabsmark = <number of (tangible) absorbing markings>;
_nvanmark = <number of vanishing markings>;
_nvanloop = <number of transient loops>;
_nentries = <number of arcs in the reachability graph>;
_reachset =
   <mk><lbl>      <pl>:<tk>  ...  <pl>:<tk>;
   ....................................
   <mk><lbl>      <pl>:<tk>  ...  <pl>:<tk>;
_reachgraph =
   <mk>      <mk>:<tr>:<val>  ...  <mk>:<tr>:<val>;
   ....................................
   <mk>      <mk>:<tr>:<val>  ...  <mk>:<tr>:<val>;
```

where <mk> is the integer index of a marking (non-negative for tangible markings, negative for vanishing markings) and <lbl> is a code (_T for tangible, non-absorbing; _A for tangible, absorbing; _V for vanishing marking not in a loop; _L for vanishing marking in a transient loop); <pl> is the non-negative integer internally assigned to each place (in the same order of definition in the CSPL file); <tr> is the non-negative integer internally assigned to each transition (in the same order of definition in the CSPL file); <tk> is the (positive) number of tokens in a place; and <val> is the transition rate or probability in the marking. So, for example, this row in the reachability set specification

```
3_A    0:1    6:5;
```

means that marking 3, an absorbing tangible marking, has one token in place 0 and five tokens in place 6, while this row in the reachability graph specification

```
-4    4:2:0.7    -6:5:0.3;
```

means that marking -4, a vanishing marking, goes to marking 4 by firing transition 2 with probability 0.7, and to marking -6 by firing transition 5 with probability 0.3 (of course both transition are immediate). If the option **IOP_PR_FULL_MARK** is turned on, the format for the description of the reachability set is instead

```
_reachset =
#          <place1>   <place2>  ... <placeN>
<mk><lbl>      <tk>      <tk >  ...     <tk>
 ....................................
<mk><lbl>      <tk>      <tk >  ...     <tk>
```

8.2 The ".mc" file

This file can be generated if the option **IOP_PR_MC** is set to **VAL_YES**. If **IOP_MC** has value **VAL_CTMC**, this file describes the CTMC derived from your SRN; the vanishing markings are absent and only numerical rates appear. The format is:

```
_firstindex = 0;
_nstates = <number of states>;
_nentries = <number of arcs in the CTMC>
_order = <_FROMTO or _TOFROM>;
_matrix =
   <state>       <state>:<rate>  ...  <state>:<rate>;
```

```
   ..................................................
   <state>       <state>:<rate>  ...  <state>:<rate>;
[_initstate =
   <state>:<prob> ... <state>:<prob>
   ..........................................
   <state>:<prob> ... <state>:<prob>;]
_iterations = <maximum number of iterations>;
_precision = <requested precision>;
_method = <requested solution method>;
_time = <_INFINITY>|<time_point>;
...................
```

All entries enclosed in square brackets ([...]) are optional. Any of the statements shown after the **_matrix** description can repeat. The transition rate matrix is described by rows. If **_FROMTO** is in effect,

```
   7            5:0.4       8:1.2      12:100;
```

means that the transition rate from state 7 to state 5 is 0.4, to state 8 is 1.2, to state 12 is 100.0. The first index is 0, so if the number of states is 15, they will be identified as 0,1,2,...,14. If the order is **_TOFROM**, the transpose of the transition rate matrix will be printed. In our example, there will be rows

```
   5            ...  7:0.4  ...;
   8            ...  7:1.2  ...;
   12           ...  7:100  ...;
```

If **IOP_MC** has value **VAL_DTMC**, this file describes the DTMC derived from your SRN, the vanishing markings are still present and probabilities are given instead of rates (the matrix is stochastic).

8.3 The ".prb" file

This file describes the transient and steady-state probability for each tangible marking; it corresponds to the result of the CTMC solution (even when the actual solution used a DTMC). It can be generated if the option **IOP_PR_PROB** is set to **VAL_YES**. The format is as following:

```
_firstindex = 0;
```

```
_nstates = <same value as in input>;
_method = <method actually used>;
_precision = <the reached precision>;
_iterations = <the actual number of iterations>;
_time = <_INFINITY>|<time_point>;
_probabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;
[_derprobabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;]
[_cumprobabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;]
[_dercumprobabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;]
```

Chapter 9

User guide for iSPN

9.1 Introduction

9.1.1 Organization of this guide

The guide assumes the user already has some experience with SPNP. It is intended for first time users of the integrated Stochastic Petri Net Package (iSPN).

9.1.2 Conventions used in this chapter

The following conventions are followed in the manual

1. All buttons are displayed in boldface.
2. A click is always left mouse button unless explicitly mentioned otherwise.
3. LMB: left mouse button; MMB: middle mouse button; RMB: right mouse button.

9.2 iSPN

9.2.1 Why iSPN?

Interaction with computers has come a long way since the arcaic textual interfaces. There is now substantial literature on **human-computer interaction** (HCI), a research subject widely recognized as a vital component of successful computer applications . But, when we evaluate HCIs currently available on analytical modeling packages, we see the enormous gap between their interfaces and modern HCI trends. The developers of analytic modeling packages need to deliver beneficial services to the user, and del iver them in a usable way. This paper suggests an approach of delivering this next generation of modeling tools with improved HCI. The approach is followed in the development of an integrated environment for modeling using Stochastic Petri Nets, named iSPN. Careful consideration

was given to the design and implementation of iSPN to facilitate the creation of SPN models. iSPN increases the power of SPNP (the Stochastic Petri Net Package) by providing a means of rapidly developing stochastic reward nets (SRNs); the model type used for input. Input to SPNP is specified using CSPL (C based SPN Language), but iSPN removes this burden from the user by providing an interface for graphical representation of the model.

The development uses the scripting language Tcl (Tool Command Language), developed by Prof. John Ousterhout of U.C. Berkeley, and extension Tk, a toolkit for X windows. The selection of the script-based approach is due to three of its main benefits:

- Tcl/Tk provides a higher-level interface to X than most standard C library toolkits.
- The user interface is clearly isolated from the rest of the application, making the overall design easy to maintain and expand.
- The use of Tcl/Tk makes this application portable to all platforms.

9.3 iSPN interface

iSPN mimics the look and feel of real file cabinet drawers by using *windows with title tabs* to ease the sorting of overlapping windows. The design model of iSPN is : (i) *user-centered* and involve users as much as possible so that they can influence it; (ii) *integrates* knowledge and expertise from the different disciplines that contribute to HCI design; and (iii) be highly *iterative* so that testing can be done to check that the design does indeed meet users' requirement.

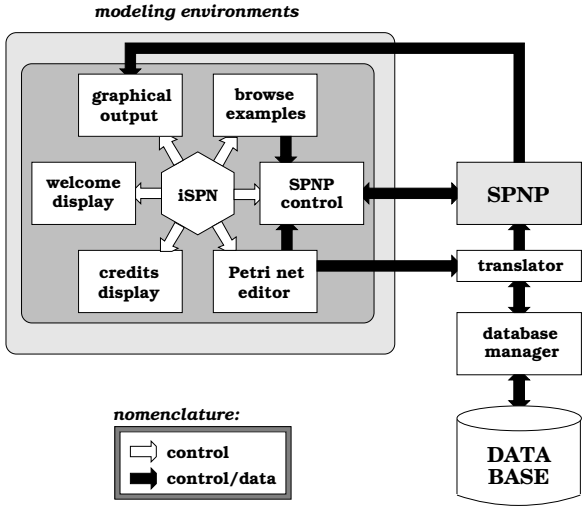


Figure 9.1: iSPN main software modules.

iSPN opens with the **WELCOME** page. The **Center for Advanced Computing and Communication (CACC)** logo is displayed. See Appendix for more information on CACC.

The major components of the iSPN interface (see Figure 9.1) are a Petri net editor which allows graphical input of the stochastic Petri nets and an extensive collection of visualization routines to analyze output results of SPNP and aid for debugging. Each module corresponds to a page in the software.

iSPN provides a high level input format to CSPL which provides great flexibility to users. iSPN is capable of executing SPNP with two different file formats: (i) files created in the CSPL (C-based Stochastic Petri net Language) ; and (ii) files created us ing iSPN's Petri Net editor.

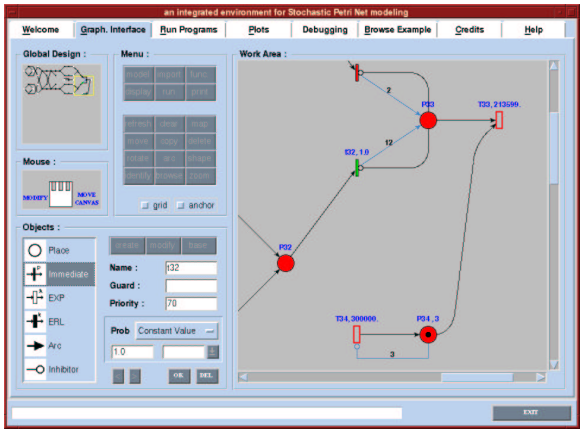


Figure 9.2: The petri net editor.

The Petri net editor, which is the software module of iSPN that allows users to graphically design the input models, introduces a new way of programming SPNP: the user can draw the SRN model and establish all the necessary additional functions (i.e., rewards rates, guard function, etc.) through a common environment. The Petri Net editor provides several characteristics normally available only in sophisticated two-dimensional graphical editors and a lot of new features designed specifically for the SPNP environment.

iSPN also provides a textual interface, through the **Run Programs** page (see Figure 9.3), which is necessary if we wish to accommodate several categories of users. The user can select his model, edit it, and execute it, without having to switch between different appplications like a text editor and SPNP. Beginners may feel more comfortable using the Petri Net editor whereas experienced SPNP users may wish to input their models using CSPL directly. Even if the textual input is the option of choice, a lot of new facilities are offered through the integrated environment. In both cases, the "SPNP control" module provides everything a user needs to run and control the execution of SPNP without having to switch back-and-forth among distinct environments (i.e., the UNIX command or a text

editor).

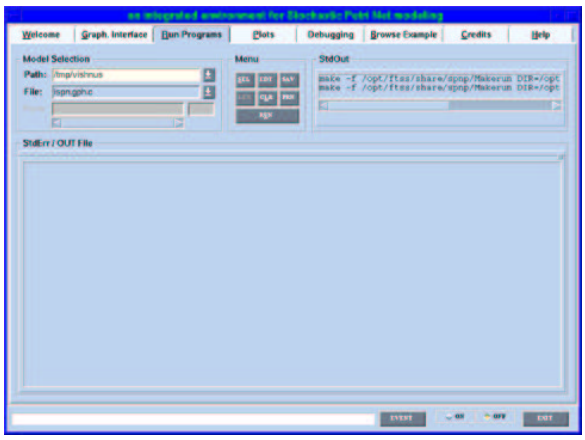


Figure 9.3: The execution page.

In the **Plots** page, the user can view SPNP's results in the GUI application. iSPN's own graphing capability allows the results of experiments to be graphically displayed in the same environment. Different combinations of input data may be compared against each other on one plot or viewed simultaneously. The graphical output format is created in such a way that it may be viewed by other vizualization packages, such as gnuplot or xvgr.

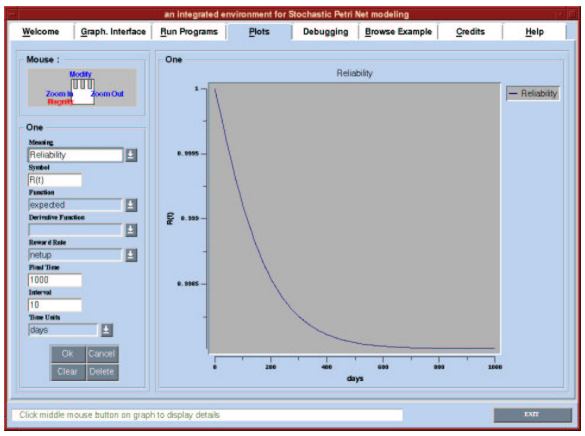


Figure 9.4: The output page.

Previously, debugging was a difficult task due to the textual environment. Opening many intermediate files created during the compilation of SPNP was necessary in order to know the validity of the CSPL file. One of these intermediate files, ".rg", is important for finding bugs in the model description, if they exist. This file is a description of the reachibility graph and is displayed in a tree format as a part of iSPN. The **debugging** feature is based on navigation of the reachability graph result ing in an innovative function of iSPN that should provide improved efficiency in the development of stochastic Petri net models.

iSPN also offers the user the unique ability to efficiently and selectively **browse a database of old SPNP models**. This specialized browser provides an easy way of organizing SPNP programs, which can be very beneficial, even for the more seasoned use rs.

The following sections describe how to use these capabilities of iSPN.

9.4 The Petri net editor

The Petri net editor handles the creation of the SPN. The **WORK AREA**, which constitutes the major portion of the screen, is where the user designs his stochastic Petri net. Icons of the various objects used in SPN design appear in the **Objects menu** (see Figure 9.5).

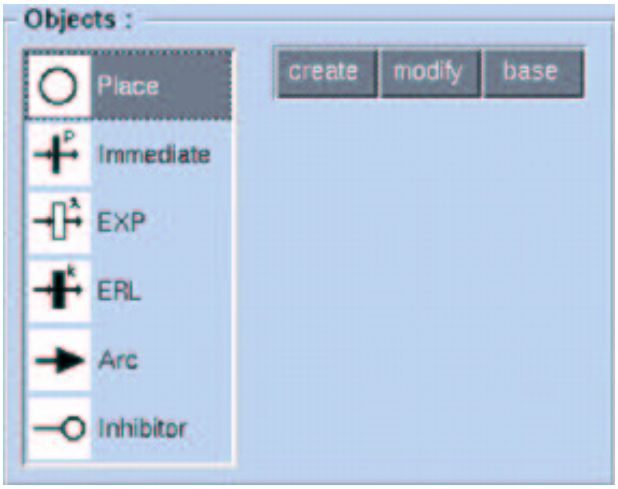


Figure 9.5: The objects menu.

At all times the **mouse bindings** are displayed above the Objects menu. For large PNs which extend beyond the bounds of the Work Area, the user can get a global view of the design in the **Global Design** window (see Figure 9.6).

iSPN provides numerous functions to simplify design of Petri nets (see Figure 9.7).

Creation of the SRN

To create an SPN,

1. Click on **Create**.
2. Select object.

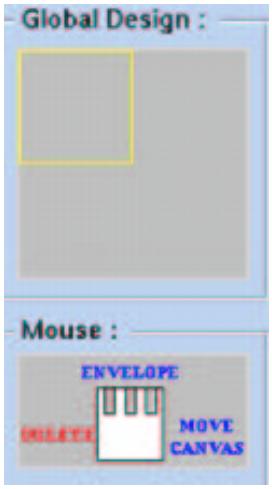


Figure 9.6: The global view.

Figure 9.7: The iSPN menu.

3. Move the cursor into the Work Area and click. The object appears in green and a table appears next to the Work Area for the object parameters.

4. Press Return to confirm operation.

5. Repeat for each object.

SPN modification

Features for SPN modification can be broadly categorized into

1. Parameter modification

2. Physical object modification

Parameter modification involves altering the parameters already assigned to a specific object. To change the value of an object

1. Click **Modify**
2. Select the object in the Work Area.
3. Modify the attributes of the object in the table.

Physical object modification deals with changing the position and shape of the objects. iSPN provides the following means for object modification :

Copy

1. Click on **Copy**
2. Select the area to be copied from the Work Area.
3. Click to acknowledge.
4. Move the cursor to a different location in the Work Area and click. The object is automatically copied.

Move

1. Click **Move**
2. Use the LMB to select a single object to move or the MMB for a group of objects.
3. Move cursor to the desired position and click to place object/group.

Delete

1. Click **Delete**
2. Use the LMB to select a single object to move or the MMB for a group of objects.
3. Double LMB click to delete

Rotate : This is used to rotate the transitions for a clearer and neater model. However a rotated transition does not change the functionality of the model in any way!!

1. Click **Rotate**
2. Click on the transition to rotate. Each click rotates the transition by 45 degrees in the clockwise direction .

Arc : This function changes the shape of the arc. It comes in handy when an arc intersects other objects. By clicking on intermediate points on the arc, it can be re-routed so that it does not cross over objects.

To re-route the arc,

1. Click on **Arc**
2. Click on the arc with the MMB, and while holding the button down reposition the arc.

When the arc is repositioned, it consists of a number of line segments. This can be changed by using the **Shape** function.

1. Click **Shape**
2. Click on the intermediate points to smoothen out the vertices.

9.4.1 File functions

All file functions, like save, open, etc, are accessed through the **model** button. When the **model** button is clicked, the Work Area is replaced by a screen with various file options. Brief descriptions of the file functions are given below.

1. **New** : Create a new worksheet.
2. **Open** : Open an existing worksheet.
3. **Save** : Save the current worksheet.
4. **Save As** : Save the current worksheet under a different name.

The user has the option of saving the worksheet with information such as Project Name, Version, Reference, and Date Last Modified. There is also a Comment Area where the user can provide a brief description of the Petri Net.

9.4.2 Miscellaneous functions

A number of other functions are provided for ease of design. These functions are

Zoom : Zooms in to/out of worksheet.

Map : In cases where the Petri net does not fit in the Global Work Area, the **map** function is used to cut/add space to the design sheet.

To add space,

1. Click **Map**.
2. Doubleclick on the Work Area with the MMB.
3. Each doubleclick adds space all around the Work Area.

To delete space,

1. Click **Map**
2. Doubleclick on the Work Area with the LMB.
3. Each double click deletes space all around the Work Area.

Note that initial design space is the minimum available. Space cannot be deleted without first adding space.

Refresh : Redraw worksheet

Clear : Clear worksheet

To clear worksheet,

1. Click **Clear**
2. Doubleclick in the Work Area with the LMB.

Note : Use this with caution!!! However, as a protection against errors, the entire worksheet can be restored to its original condition

To restore the worksheet,

1. Click **Clear**

2. Doubleclick in the Work Area with MMB.

Grid : Draws a grid on the worksheet.

Anchor : *snap* object to the closest grid point.

9.4.3 Environment control functions

An interesting feature of SPNP is its flexibility of operation, allowing the user to tailor its functioning to his/her own needs. It allows the user to change the operating environment. This is done in iSPN by

1. Click **model**.
2. Click **environment**.
3. Select features.

9.4.4 Information functions

Information functions allow the user to obtain information about his/her design. iSPN provides a number of ways to view information.

Identify : Display information about the object over which the mouse pointer is positioned.

Browse : Display information about all objects in a tabular form.

1. When an object is selected from the table, it is highlighted in green in the Global View.
2. After selection, when **OK** is clicked, the Work Area is restored with the selected object highlighted in green.

Display : Allow user to select the information he/she wants to have displayed on the screen, next to the objects.

9.4.5 FSPN model

In SPNP v6, a very powerful function is introduced. That is Fluid Stochastic Petri Net (FSPN). Fluid places and arcs are introduced into ordinary SPNs to represent continuous quantities. iSPN provides the editing environment for FSPN.

In FSPN, the fluid places are drawn as two concentrical circles and the fluid arcs are drawn as double lined arrows, to differentiate with the single circle representation of discrete places and single lined arrow representation of discrete arcs. In iSPN editing, for each places and arcs, there is an attribute checkbox for fluid places and fluid arcs. Clicking on this attribute checkbox will change the corresponding place/arc into fluid place/arc and verse vesa. iSPN will adjust the graphics output according to these attributes. Several options have to be set for FSPN model (Refer to Section 7). The execution of FSPN model is through simulation, i. e., IOP_SIMULATION should be set to VAL_YES before execution. The results will be displayed in the output panel after the simulation.

9.5 Execution of the model

The design can be executed from two places. One is from the Petri net editor itself, and the other is from the Run Programs page.

From the Petri Net Editor page,

1. Click **Run**.
2. The Run Programs page is pulled up, and the current model is executed.
3. If the model has been incorrectly specified, a list of errors is displayed.
4. If there are no errors, an OK message is signalled.
5. Click **Mapping**. The CSPL source code is generated and displayed.
6. To control the execution environment of SPNP click **Environment**.
7. Click **Execute** to run the model.
8. After the model is executed, the CSPL code is shown again. To view the output, switch to the **Run Programs** page.
9. To view the **StdErr** files, pull the top edge of the message window in the **Run Programs** page down.

10. Other files, like the Log and the Reachability Graph files, can be displayed by clicking the RMB anywhere in the message window.

The **Run Programs** page has **model selection** information (path and filename of project), a menu for **file, view/edit, and execute** operations, a message window for **StdOut** (all output message) and a message window for **StdErr** (error messages) (see Figures 9.8, 9.9).

Figure 9.8: Run Programs menu

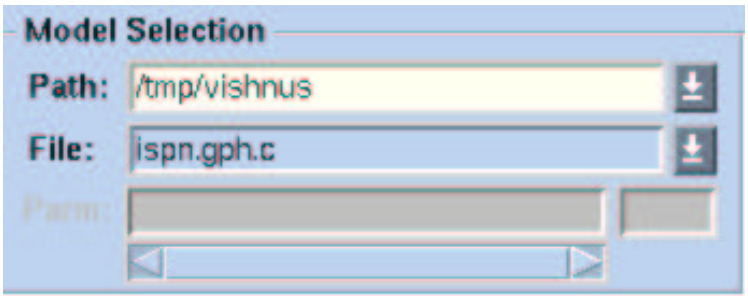


Figure 9.9: Model Selection menu

From the **Run Programs** page,

1. Click **Sel** to select model to be run.
2. Click **Run** to run the model.
3. Click **Edt** to view/edit the CSPL file for the selected model. When **Edt** is clicked, the emacs editor is invoked.
4. Click **Sav** to save the .C, .SPN, .log, .rg, .out files
5. Click **Clr** to clear all messages in StdOut.

9.6 Viewing output

Output can be observed in two ways - one is textual, as was described in the previous section, and the other way is graphical, in the form of line graphs. Graphs are viewed in the **Plots** page. The **Plots** page opens with a **menu** for the graphing functions, **mouse bindings**, **graph definition**, and **model selection** areas.

Figure 9.10: Plots model selection menu

The **model selection** (see Figure 9.10)area handles various file functions.

Open : Open an existing .isp file for execution.

Save : Save the current .isp file or the .igp (graphical output) file.

Save As : Save the current file under a different name.

Environment : Change operating environment of SPNP.

Once a .isp file has been opened, to create a graph,

1. Click **Create**.
2. Assign meanings to the graphs, and the symbol which must appear on the graph.
3. Define the function, derivative function (if any), and reward rates for the graph.
4. Input the time of execution, the time interval, and the time unit.
5. Click **OK** if all is well, **Cancel** to cancel, and **Clear** to clear all fields.
6. When **OK** is clicked, **Run** highlights in green if all the fields have been correctly entered.
7. Click **Run**
8. If there are no syntax errors in the file, an OK message is signalled. If there are errors, a list of errors is displayed.
9. Execution environment of SPNP can be modified through the **Environment** button.

10. Click **Mapping**. The CSPL file is displayed.
11. Click **Execution**. The **Run Programs** page is brought up, and the model executed.
12. After model execution, the graphical output is displayed.

9.6.1 Graphing functions

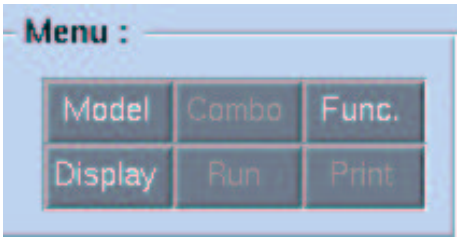


Figure 9.11: Graphing functions

Combo : Used to combine two or more graphs in the same display window. A maximum of four graphs can be displayed at any given time in any given display window.

1. Click **Combo**.
2. A combine window is displayed beneath the Menu. Select the graphs to be combined and click **OK**.
3. The selected graphs are displayed in the display window.

Display : controls the display window parameters. The user can control the scale, crosshairs, legend, and the grid.

1. Click **Display**.
2. Select options for each display window.
3. Click **OK**.

Func

Print : Prints the output to printer.

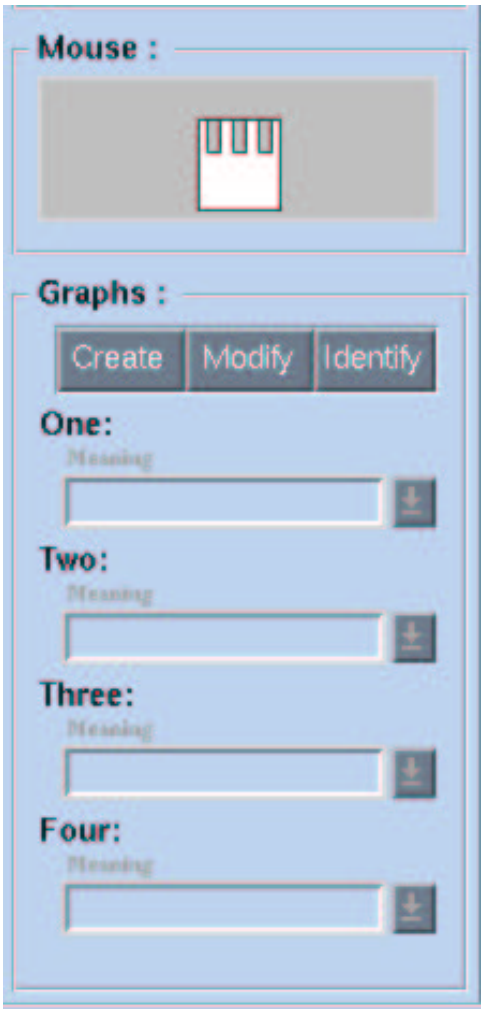


Figure 9.12: The Plots menu

9.6.2 Graph definition functions

Modify : Used to modify the graph definitions.

1. Click **Modify**
2. Select the graph to be modified.
3. Make the changes in the **Modification Area**.
4. Click **OK**.
5. For the changes to take effect, the model must be rerun. Click **Run**.

Identify

This is used to identify the graphs.

1. Click **Identify**.
2. Click with the MMB on any of the display windows. The graph details are displayed below the mouse bindings.

Note : Positioning the mouse over any point of the graph will display the value at that point.

9.7 Debugging

This page, as the name implies, is used for debugging purposes. The reachability graph plays a very important role in debugging a model.

The general layout of the page is as follows : A major part of the window is dedicated to the reachability graph. On the left of the reachability graph area are buttons for **opening** .rg files, and displaying information about the reachability graph. There are also options for zooming in/out and identification of repeated markings.

A brief description of the usage of the page is given below :

OPEN

The **OPEN** button allows the user to open **.rg** files. Once a .rg file has been opened, the reachability graph is graphically displayed in the window.

INFO

The **INFO** button provides the user with the salient and important information about the reachability graph.

Place and Transition Areas

These are present in order to identify the various places and transitions presented in the PN model.

Zoom In/Out

This allows the user to zoom in or out of the reachability graph.

Identify

When **Identify** is on, moving the mouse pointer over a marking highlights all repetitions of the marking.

9.7.1 Reachability graph traversal

The markings of the reachability graph are displayed in cyan, with the current marking in green. The graph can be traversed either by using the cursor keys or the mouse. The user can also view the changes in the PN model. This is done in the following way :

1. Click **Open**.
2. Select the .rg file.
3. Click **OK**.
4. Press **F2** on the keyboard.
5. A message box appears asking whether the user wishes to open a .isp file.
6. Click **OK**.
7. Select the .isp file corresponding to the .rg file.
8. The PN model appears in the window.
9. The user can switch between the Petri net and the reachability graph by using the **F2** and **F3** keys.

9.8 Browse examples

The **Browse Examples** page allows the user to browse through a database of examples. All the user needs to do is

1. Select a file from the left side of the window.
2. The CSPL code for the file is displayed.
3. Click **Run** to execute the model.

9.9 Help

The following are the ways in which iSPN provides on-line help for the user.

1. iSPN provides hyperlink help for its users. The sixth "page" of iSPN is the **Help** page, which gives detailed information about the various functions in the pages.
2. Balloon help for all pages. If the mouse is positioned over a button for a certain amount of time, a window pops up giving a brief description of the button.
3. F1 help. If F1 is pressed with the mouse positioned over a button, hypertext help is pulled up for the button.
4. Mouse bindings. At all times the mouse bindings are displayed. Adjacent to the mouse buttons are messages telling the user what function the corresponding button carries out. iSPN follows a color convention. A blue message corresponds to a single click, a red one to a double click, and a green one to a 'Ctrl' click.
5. Message bar. When any operation is carried out, a message appears at the bottom of the iSPN window informing the user of the status of the operation. Again, a color convention is followed. Messages in green are OK messages, while those in red mean that there is an error.

9.10 How to install iSPN in a unix environment

A version of Tcl/Tk with their extensions is distributed with the package in the directory tcltk. If you can't run the application, remove the directory tcltk, then install it in the same path.

You need to install in the order if you have any problems:

1. **Tcl**
   Home page : http://www.sco.com/Technology/tcl/Tcl.html
   Download the code : http://sunscript.sun.com/TclTkCore/index.html

2. **Tk**
   Home page : http://www.sco.com/Technology/tcl/Tcl.html
   Download the code : http://sunscript.sun.com/TclTkCore/index.html

3. **Tix**
   Home page : http://www.xpi.com/tix/
   Download the code : http://www.xpi.com/download/binaries.html

4. **Blt**
   Download the code : http://www.sco.com/Technology/tcl/Tcl.html
   ftp://ftp.neosoft.com/pub/tcl/alcatel/extensions/

5. **Expect**
   Home page : http://expect.nist.gov/
   Download the code : http://expect.nist.gov/

6. **Tcl-my-fancy**
   Comprehensive interpreter for tcl/tk/tix/blt/expect commands
   Download the code : http://www.nsrc.nus.sg/STAFF/rthien/d_bugger/mps.cap

Type in your shell the script file “script” after you have modified it to correspond with the directory where you have installed the packages iSPN and SPNP.

1. setenv TIX_LIBRARY dir0
   dir0 : directory contained the library files for tix examples: /opt/tcltk/lib/tix

2. setenv ISPN2_DIRECTORY dir1
   dir1 : directory contained the ispn code.

3. setenv SPNP_DIRECTORY dir2
   dir2 : directory contained the SPNP executable.

Operation in “ispn” file : Change the first line and write the correct path of the global interpreter.

9.11 Programming resources

The selection of the script-based approach is due to three of its main benefits: Tcl/Tk provides a higher-level interface to X than most standard C library toolkits. Development of the IDEAS environment will be fast because of fast turnaround, aiding the debbuging process and refinement of the interface. The user interface is clearly isolated from the rest of the application, making the overall design easy to maintain and expand.

9.11.1 Tcl (version tcl7.4)

Tcl stands for ”Tool Command Language”. Tcl is really two things: a scripting language, and an interpreter for that language. Tcl was designed and crafted by Prof. John Ousterhout of U.C Berkeley. Tcl can be used in commercial applications for free. The interpreter has been ported from UNIX to DOS and Macintosh environments. As a scripting language, Tcl is similar to other UNIX shell languages which let you execute other programs. Tcl provides enough programmability (variables, control flow, procedures) that we can build up complex scripts that assemble existing programs into a new tool tailored to our needs.

9.11.2 Tk (version tk4.0)

Tk is a toolkit for window programming. It was designed for the X window system. Tk shares many concepts with other windowing toolkits. Tk provides a set of Tcl commands that create and manipulate widgets. A widget is a window in a graphical user interface that has a particular appearance and behavior. Widget types include buttons, scrollbars, menus, and text windows. Tk also has a general purpose drawing widget called a canvas that lets you create lighter-weight items such as lines, boxes and bitmaps. The X window system supports a hierarchy of windows, and this is reflected by the Tk commands, too. To an application, the window hierarchy means that there is a primary window, and then inside that window there can be a number of children windows. The children windows can contain more windows, and so on. Just as a hierarchical file system has directories that are containers for files and directories, a hierarchical window system uses windows as containers for other windows. The hierarchical affects the naming scheme used for Tk widgets as described below, and it is used to help arrange widgets on the screen. Widgets are under the control of a geometry manager that controls their size and location on the screen. Until a geometry manager learns about a widget, it will not be mapped onto the screen and you will not see it. The main trick with any geometry manager is that you use frame widgets as containers for other widgets. One or more widgets are created and then arranged in a frame by a geometry manager. A Tk-based application has an event driven control flow, just as with most window system toolkits. An event is handled by

associating a Tcl command to that event using the bind command. There are a large number of different events defined by the X protocol, including mouse and keyboard events. Tk widgets have default bindings so you do not have to program in detail by yourself. You can also arrange for events to occur after a specified period of time with the “after” command. Event bindings are structured into a simple hierarchy of global bindings, class bindings, and instance bindings. An example of a class is Button, which is all the button widgets. The Tk toolkit provides the default behavior for buttons as bindings on the button class. You can supplement these bindings for an individual button, or define global bindings that apply to all bindings. You can even introduce new binding classes in order to group sets of bindings together. The binding hierarchy is controlled with the bindtags command. The basic structure of a Tk script begins by creating widgets and arranging them with a geometry manager, and then binding actions to the widgets. After the interpreter processes the commands that initialize the user interface, the event loop is entered and the application begins running.

9.11.3 Tix (version Tix4.0.4)

Tix, the Tk Interface Extension, is an extensive set of over 40 mega-widgets including: ComboBox, Motif style FileSelectBox, MS Windows style FileSelectBox, PanedWindow, NoteBook, Hierarchical List, Directory Tree, File Manager and many more. Tk only provides a set of primitive widgets that may be tedious to work with. In constrast, Tix delivers powerful higher-level widgets that fit the needs of your application. With Tix, you can forget about the frivolous details of the Tk widgets and concentrate on solving your problems at hand.

Professional Look-and-feel

Tix defines configurations options that are very close to the standard Motif look-and-feel. If you like the ease of programming with Tcl/Tk but want your program to have an industrial standard look-and-feel, Tix is the answer.

Rapid Prototyping New Widgets

The Tix Intrinsics API makes it possible to write new custom designed widgets using Tcl exclusively. It typically reduces the efforts of developing a new widget by a factor of ten or more.

Chapter 10

Examples

10.1 Molloy's example

10.1.1 Source

M. K. Molloy, Performance Analysis Using Stochastic Petri Nets, *IEEE Trans. Comput.*, C-31 (9), Sept. 1982, 913–917.

10.1.2 Description

The net is shown in Figure 10.1

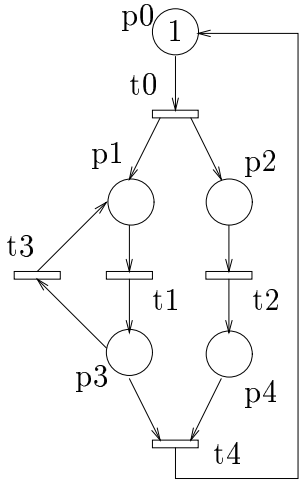


Figure 10.1: SPN for Example 10.1

7

10.1.3 Features

- Assertion on place $p3$.
- Reward based functions to compute expected values.
- Default measures
- Steady-state analysis

10.1.4 SPNP File — *example1.c*

```
/* This example adapted from M.K. Molloy's IEEE TC paper */

# include  "user.h"

void options() {

   iopt(IOP_SSMETHOD, VAL_GASEI);
   iopt(IOP_PR_FULL_MARK,VAL_YES);
   iopt(IOP_PR_MARK_ORDER,VAL_CANONIC);
   iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
   iopt(IOP_PR_MC,VAL_YES);
   iopt(IOP_PR_PROB,VAL_YES);
   iopt(IOP_MC,VAL_CTMC);
   iopt(IOP_PR_RSET,VAL_YES);
   iopt(IOP_PR_RGRAPH,VAL_YES);
   iopt(IOP_ITERATIONS,20000);
   fopt(FOP_ABS_RET_M0,0.0);
   fopt(FOP_PRECISION,0.00000001);

}

void net() {

   place("p0");
   init("p0",1);
   place("p1");
   place("p2");
   place("p3");
   place("p4");

   rateval("t0",1.0);
   rateval("t1",3.0);
   rateval("t2",7.0);
   rateval("t3",9.0);
   rateval("t4",5.0);

   iarc("t0","p0");   oarc("t0","p1");   oarc("t0","p2");
   iarc("t1","p1");   oarc("t1","p3");
   iarc("t2","p2");   oarc("t2","p4");
   iarc("t3","p3");   oarc("t3","p1");
   iarc("t4","p3");   iarc("t4","p4");   oarc("t4","p0");
}

int assert() {
   if (mark("p3") > 5)
      return(RES_ERROR);
```

```
   else
      return(RES_NOERR);
}

void ac_init() {
   fprintf(stderr,"\nExample from Molloy's Thesis\n\n");
   pr_net_info(); /* information on the net structure */
}

void ac_reach() {
   fprintf(stderr,"\nThe reachability graph has been generated\n\n");
   pr_rg_info(); /* information on the reachability graph */
}

/* general marking dependent reward functions */
double   ef0() { return((double)mark("p0")); }
double   ef1() { return((double)mark("p1")); }
double   ef2() { return(rate("t2")); }
double   ef3() { return(rate("t3")); }
double   eff() { return(rate("t1") * 1.8 + (double)mark("p3") * 0.7); }

void ac_final() {

   solve(INFINITY);

   pr_mc_info(); /* information about the Markov chain */
   pr_expected("mark(p0)",ef0);
   pr_expected("mark(p1)",ef1);
   pr_expected("rate(t2)",ef2);
   pr_expected("rate(t3)",ef3);
   pr_expected("rate(t1) * 1.8 + mark(p3) * 0.7",eff);
   pr_std_average(); /* default measures */
}
```

10.2 Software Performance Analysis

10.2.1 Description

This example models the following piece of software:

```
A: Statements;
PARBEGIN
   B1: statements;
   B2: IF (cond1) THEN
         C: statements;
       ELSE
          DO
             D: statements;
          WHILE (cond2);
```

```
            END IF
        PAREND
```

The corresponding SPN model is shown in Figure 10.2.

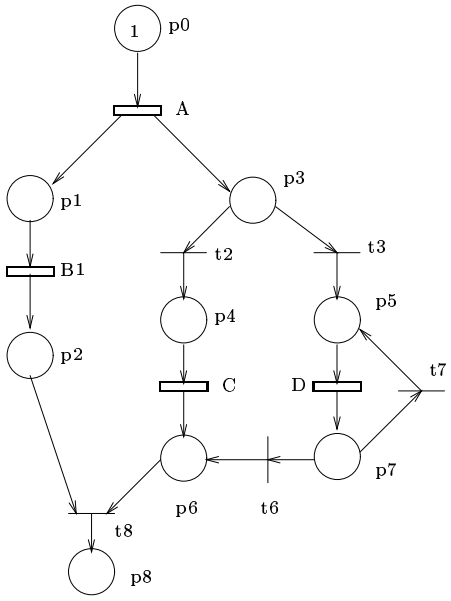


Figure 10.2: SPN for Example 10.2

10.2.2 Features

- Probability and rate functions.
- Priorities for immediate transitions.
- Reward functions.
- Transient analysis with multiple time points.

10.2.3 SPNP File — *example2.c*

```
# include   "user.h"

/*
```
This example corresponds to the following piece of software:

```
A:  statements;
   PARBEGIN
   B1:  statements;    B2:  IF cond THEN
                       C:  statements;
                       ELSE
                          DO
                          D:  statements
                          WHILE cond;
                       IFEND
   PARAEND
*/

options() {
   /* Transient analysis */
   iopt(IOP_TSMETHOD,VAL_TSUNIF);
}

/* rates and probabilities are defined as functions */
double          rate0() { return(1.0);}
double          rate1() { return(0.3);}
double          prob2() { return(0.4);}
double          prob3() { return(0.6);}
double          rate4() { return(0.2);}
double          rate5() { return(7.0);}
double          prob6() { return(0.05);}
double          prob7() { return(0.95);}
double          prob8() { return(1.0);}

net() {
   place("p0");
   init("p0",1);
   place("p1");
   place("p2");
   place("p3");
   place("p4");
   place("p5");
   place("p6");
   place("p7");
   place("p8");

   /* priorities associated with transitions */
   imm("t2");   priority("t2",1);
   imm("t3");   priority("t3",1);
   imm("t6");   priority("t6",1);
   imm("t7");   priority("t7",1);
   imm("t8");   priority("t8",1);

   /* rate and probability functions */
   ratefun("A", rate0);
   ratefun("B1",rate1);
   probfun("t2",prob2);
   probfun("t3",prob3);
   ratefun("C", rate4);
   ratefun("D", rate5);
   probfun("t6",prob6);
   probfun("t7",prob7);
   probfun("t8",prob8);

   iarc("A","p0");      oarc("A","p1");      oarc("A","p3");
   iarc("B1","p1");   oarc("B1","p2");
   iarc("t2","p3");   oarc("t2","p4");
   iarc("t3","p3");   oarc("t3","p5");
```

```
    iarc("C","p4");      oarc("C","p6");
    iarc("D","p5");      oarc("D","p7");
    iarc("t6","p7");   oarc("t6","p6");
    iarc("t7","p7");   oarc("t7","p5");
    iarc("t8","p2");    iarc("t8","p6");   oarc("t8","p8");
}

assert() { return(RES_NOERR); }

ac_init() { fprintf(stderr,"\nSoftware modeling example\n\n"); }

ac_reach() { }

double rfunc() { return(mark("p8")); }

ac_final() {
  int i;

   /* Transient analysis with multiple time points */
   /* reward function */
  for (  i = 1; i < 10; i++ )
  {
    solve( (double) i );
    pr_expected("probability of completion", rfunc);
  }

  for (  i = 10; i <= 20; i += 2 )
  {
    solve( (double) i );
    pr_expected("probability of completion", rfunc);
  }

  for (  i = 20; i <= 50; i += 5 )
  {
    solve( (double) i );
    pr_expected("probability of completion", rfunc);
  }

}
```

10.3 $M/M/m/b$ queue

10.3.1 Description

This example models a finite-buffer $M/M/m/b$ queue shown in Figure 10.3. The corresponding SPN is shown in Figure 10.4.

10.3.2 Features

- Both steady-state and transient analysis.

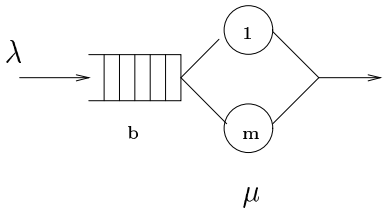
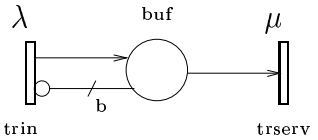


Figure 10.3: The $M/M/m/b$ Queue.



Transition	Rate Function
trserv	$\#(buf)\mu$ if $(\#(buf) < m)$ $m\mu$ otherwise

Figure 10.4: SPN for Example 10.3

- Marking dependent firing rates.
- Assertions.
- General reward specification.

10.3.3 SPNP File — *example3.c*

```
/* This example models a Multi-server FCFS queue with finite buffer */
/* An M/M/m/b queue */

# include   "user.h"

   /* Global variables */
   double lambda;
   double mu;
   int b;
   int m;
   int method;

void options() {

   method = input("Input 0/1 for Steady-state/Transient analysis");
   if ( method == 0 )
      iopt(IOP_SSMETHOD,VAL_SSSOR);
   else if ( method == 1 )
      iopt(IOP_TSMETHOD,VAL_TSUNIF);
   else
   {
```

```
        fprintf(stderr,"ERROR: Illegal method specification");
        exit(1);
    }

    iopt(IOP_PR_FULL_MARK,VAL_YES);
    iopt(IOP_PR_MARK_ORDER,VAL_CANONIC);
    iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
    iopt(IOP_PR_MC,VAL_YES);
    iopt(IOP_MC,VAL_CTMC);
    iopt(IOP_PR_PROB,VAL_YES);
    iopt(IOP_PR_RSET,VAL_YES);
    iopt(IOP_PR_RGRAPH,VAL_YES);
    iopt(IOP_ITERATIONS,20000);
    iopt(IOP_CUMULATIVE,VAL_NO);
    fopt(FOP_ABS_RET_M0,0.0);
    fopt(FOP_PRECISION,0.00000001);
    lambda = finput("Enter lambda");
    mu = finput("Enter mu");
    b = input("Enter the number of buffers");
    m = input("Enter the number of servers");

}

/* Marking dependent firing rate */
double rate_serv() { if ( mark("buf") < m ) return( mark("buf")*mu);
                     else return(m*mu); }

void net() {

    place("buf");


    rateval("trin",lambda);
    ratefun("trserv",rate_serv);

    oarc("trin","buf"); mharc("trin","buf",b);
    iarc("trserv","buf");
}

int assert() {
    /* Make sure that the number of tokens in buf does not exceed the
       buffer size */
    if ( mark("buf") > b )
        return(RES_ERROR);
    else
        return(RES_NOERR);
}

void ac_init() {
    fprintf(stderr,"A model of the M/M/m/b Queue");
    pr_net_info();
}

void ac_reach() {
    pr_rg_info();
}

double qlength() { return(mark("buf")); }
double util() { return(enabled("trserv")); }
double tput() { return(rate("trserv")); }
double probrej() { if ( mark("buf") == b ) return(1.0);
```

```
                else return(0.0); }

double probempty() { if ( mark("buf") == 0 ) return(1.0);
                else return(0.0); }
double probhalffull() { if ( mark("buf") == b/2 ) return(1.0);
                else return(0.0); }
void ac_final() {

   double time_pt;

   /* measures related to the queue */
   if ( method == 0 )
   {
    solve(INFINITY);
    pr_expected("Average Queue Length", qlength);
    pr_expected("Average Throughput", tput);
    pr_expected("Utilization", util);

    /* this case corresponds to buf having b tokens */
    pr_expected("Probability of rejection",probrej);

    /* this case corresponds to buf having zero tokens */
    pr_expected("Probability that queue is empty",probempty);

    /* this case corresponds to buf having b/2 tokens */
    pr_expected("Probability that queue is half full",probhalffull);
   }
   else
   {
      for ( time_pt = 0.1; time_pt < 1.0; time_pt += 0.1)
      {
         solve(time_pt);
         pr_expected("Average Queue Length", qlength);
         pr_expected("Average Throughput", tput);
         pr_expected("Utilization", util);


         /* this case corresponds to buf having b tokens */
         pr_expected("Probability of rejection",probrej);

         /* this case corresponds to buf having zero tokens */
         pr_expected("Probability that queue is empty",probempty);

         /* this case corresponds to buf having b/2 tokens */
         pr_expected("Probability that queue is half full",probhalffull);
      }
      for ( time_pt = 1.0; time_pt < 10.0; time_pt += 1.0)
      {
         solve(time_pt);
         pr_expected("Average Queue Length", qlength);
         pr_expected("Average Throughput", tput);
         pr_expected("Utilization", util);


         /* this case corresponds to buf having b tokens */
         pr_expected("Probability of rejection",probrej);

         /* this case corresponds to buf having zero tokens */
         pr_expected("Probability that queue is empty",probempty);

         /* this case corresponds to buf having b/2 tokens */
         pr_expected("Probability that queue is half full",probhalffull);
```

```
    }
  }
}
```

10.4 C.mmp system performability analysis

10.4.1 Source

J. T. Blake, A. L. Reibman and K. S. Trivedi, Sensitivity Analysis of Reliability and Performability Measures for Multiprocessor Systems, *Proc. 1988 ACM SIGMETRICS*, Santa Fe, NM, 1988.

10.4.2 Description

This example models the C.mmp system designed at CMU. The architecture of the system is shown in Figure 10.5. The corresponding SPN model is shown in Figure 10.6.

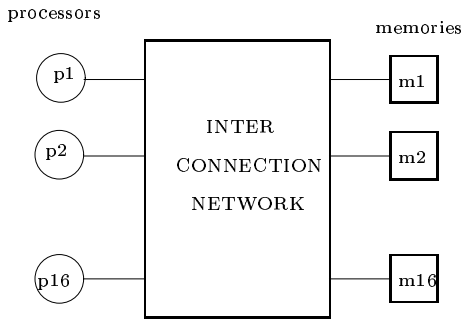
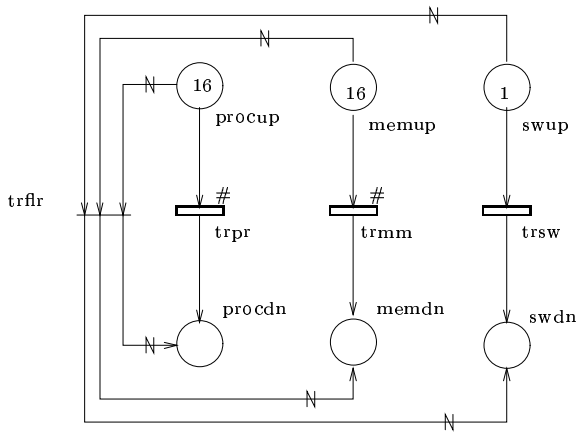


Figure 10.5: The C.mmp Architecture.

10.4.3 Features

- Enabling functions.
- Variable multiplicity arcs.
- Reward based measures.
- Transient analysis.



Transition	Enabling Function
trflr	$((\#(procup) < k) \vee (\#(memup) < k) \vee (\#(swup) = 0))$ $\wedge((\#(procup) > 0) \vee (\#(memup) > 0) \vee (\#(swup) > 0))$

Arcs	Multiplicity Function
procup $\rightarrow$ trflr & trflr $\rightarrow$ procdn	$\#(procup)$
memup $\rightarrow$ trflr & trflr $\rightarrow$ memdn	$\#(memup)$
swup $\rightarrow$ trflr & trflr $\rightarrow$ swdn	$\#(swup)$

Figure 10.6: SPN for Example 10.4.

10.4.4 SPNP File — *example4.c*

```
#include "user.h"
#include <math.h>

/*
   This is a model of the C.MMP multiprocessor system adopted from Blake,
   Reibman and Trivedi "Sensitivity Analysis of Reliability and
   Performability Measures for Multiprocessor Systems", ACM SIGMETRICS
1988.
*/

int k;

extern int abs();
extern double pow();

void options()
{
  iopt(IOP_SSMETHOD, VAL_SSSOR);
  iopt(IOP_TSMETHOD,VAL_TSUNIF);
  iopt(IOP_OK_TRANS_M0, VAL_YES);
  iopt(IOP_MC,VAL_CTMC);
  iopt(IOP_ITERATIONS,20000);
  fopt(FOP_ABS_RET_M0,0.0);
  fopt(FOP_PRECISION,0.00000001);
```

```
 k = input("Input minimum number of proc/mem needed (1<=k<=16)");
 if ( k < 1 )
   {
    fprintf(stderr,"ERROR: atleast one processor is needed (k >= 1)");
    exit(1);
   }
 if ( k > 16 )
   {
    fprintf(stderr,"ERROR: only 16 processors are available (k<=16)");
    exit(1);
   }
}

int entrflr()
{
 if ( mark("procup") == 0 && mark("memup") == 0 && mark("swup") == 0 )
   return(0);
 if ( mark("procup") < k || mark("memup") < k || mark("swup") == 0 )
   return(1);
 else
   return(0);
}

int apfl()
{
 return( mark("procup"));
}

int amfl()
{
 return(mark("memup"));
}

int asfl()
{
 return(mark("swup"));
}

void net() {
 place("procup");
 init("procup", 16);
 place("procdn");
 place("memup");
 init("memup", 16);
 place("memdn");
 place("swup");
 init("swup",1);
 place("swdn");

 /* timed transition */
 ratedep("trpr", 0.0000689, "procup");
 ratedep("trmm", 0.000224, "memup");
 rateval("trsw", 0.0002202);

 /* immediate transition */
 imm("trflr"); priority("trflr", 100); guard("trflr", entrflr);
 probval("trflr", 1.0);

 iarc("trpr", "procup"); oarc("trpr", "procdn");
 iarc("trmm", "memup"); oarc("trmm", "memdn");
 iarc("trsw", "swup"); oarc("trsw", "swdn");
```

```
  viarc("trflr", "procup", apfl); voarc("trflr", "procdn", apfl);
  viarc("trflr", "memup", amfl); voarc("trflr", "memdn", amfl);
  viarc("trflr", "swup", asfl); voarc("trflr", "swdn", asfl);

}

int assert()
{
  return(RES_NOERR);
}

void ac_init()
{
}

void ac_reach()
{
  pr_rg_info();
}

double reliab()
{
  if ( mark("procup") ≥ k && mark("memup") ≥ k && mark("swup") == 1 )
    return(1.0);
  else
    return(0.0);
}

double reward_rate()
{
  double m, l, temp;

  if ( mark("procup") ≥ k && mark("memup") ≥ k && mark("swup") == 1 )
    {
     l = min((double)mark("procup"), (double)mark("memup"));
     m = max((double)mark("procup"), (double)mark("memup"));
     temp = pow( (1.0 − (1.0 / m)) , l );
     return( m ∗ (1.0 − temp) );
    }
  else
    return(0);
}

void ac_final()
{
  double time_pt;

  for ( time_pt = 500.0; time_pt < 5000.0; time_pt += 500.0 )
    {
     solve( time_pt );
     pr_expected("Reliability", reliab);
     pr_expected("Expected Reward", reward_rate);
     pr_cum_expected("Expected Accumulated Reward", reward_rate);
    }
}
```

10.5 Database system availability analysis

10.5.1 Source

P. Hiedelberger and A. Goyal, Sensitivity Analysis of Continuous Time Markov chains using Uniformization, *Computer Performance and Reliability*, G. Iazeolla, P. J. Courtois and O. J. Boxma (Eds.), Elsevier Science Publishers, B.V. (North-Holland), Amsterdam, 1988.

10.5.2 Description

This example is a model of a database system shown in Figure 10.7.

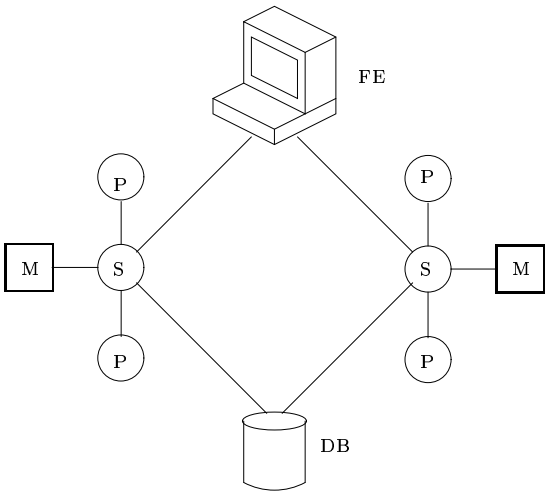
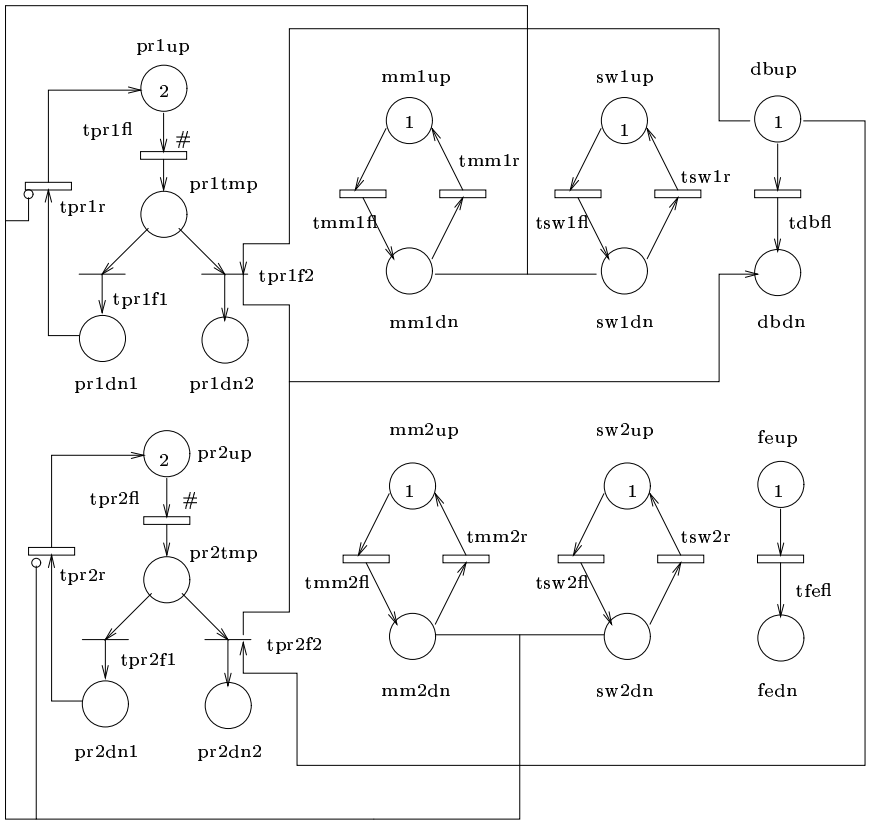


Figure 10.7: The Database System Architecture.

The system consists of a front end (FE), a database (DB) and two processing sub-systems. Each processing sub-system consists of two processors (P), a memory (M) and a switch (S). For the system to be functional, we need at least one of the processing sub-systems to be operational. The database and the front-end should also be operational. The processing sub-system is functional as long as the memory, the switch and at least one of the processors is functional. When a processor fails, with probability c it fails without disturbing the system. However, with probability $1 - c$ the failing processor corrupts the database causing it to fail and consequently rendering the system unoperational. The processors, memories and switches can be repaired while the system is up. The memories and



Transition	Enabling Function
all	$(\#(dbup) = 1) \wedge (\#(feup) = 1)$ $\wedge((\#(pr1up) > 0) \wedge (\#(mm1up) > 0) \wedge (\#(sw1up) > 0)$ $\vee(\#(pr2up) > 0) \wedge (\#(mm2up) > 0) \wedge (\#(sw2up) > 0))$

Figure 10.8: SPN for Example 10.5.

switches receive priority over the processors for repair. The corresponding SRN model is shown in Figure 10.8.

10.5.3 Features

- Global variables.
- Enabling function.
- Reward based functions.
- Transient analysis.

10.5.4 SPNP File — *example5.c*

```
/* This is a petri-net model of the database system example from
 the paper on sensitivity by Hiedelberger and Goyal */

# include   "user.h"

double coverage = 0.99;
int count = 0;

void options() {

   iopt(IOP_SSMETHOD,VAL_SSSOR);
   iopt(IOP_PR_MARK_ORDER,VAL_CANONIC);
   iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
   iopt(IOP_MC,VAL_CTMC);
   iopt(IOP_PR_RSET,VAL_YES);
   iopt(IOP_PR_RGRAPH,VAL_YES);
   iopt(IOP_CUMULATIVE,VAL_NO);
   fopt(FOP_PRECISION,0.00000001);

}

int enall() {

   /* if the database is failed */
   if ( mark("dbup") == 0 )
      return(0);

   /* if the front end is failed */
   if ( mark("feup") == 0 )
      return(0);

   /* if both the processing sub-systems are failed */
   if (( mark("mm1up") == 0 || mark("sw1up") == 0 ||
            mark("pr1up") == 0 ) &&
            ( mark("mm2up") == 0 || mark("sw2up") == 0 ||
            mark("pr2up") == 0 ))
      return(0);

   return(1);
}

void net() {

   /* first processing subsystem */
   place("mm1up");
   init("mm1up",1);
   place("sw1up");
   init("sw1up",1);
   place("pr1up");
   init("pr1up",2);
   place("mm1dn");
   place("sw1dn");
   place("pr1tmp");
   place("pr1dn1");
   place("pr1dn2");

   /* second processing subsystem */
   place("mm2up");
   init("mm2up",1);
   place("sw2up");
   init("sw2up",1);
```

```
place("pr2up");
init("pr2up",2);
place("mm2dn");
place("sw2dn");
place("pr2tmp");
place("pr2dn1");
place("pr2dn2");

/* database */
place("dbup");
init("dbup",1);
place("dbdn");

/* frontend */
place("feup");
init("feup",1);
place("fedn");


guard("tmm1fl",enall);
guard("tsw1fl",enall);
guard("tpr1fl",enall);

guard("tmm1r",enall);
guard("tsw1r",enall);
guard("tpr1r",enall);

guard("tmm2fl",enall);
guard("tsw2fl",enall);
guard("tpr2fl",enall);

guard("tmm2r",enall);
guard("tsw2r",enall);
guard("tpr2r",enall);

guard("tdbfl",enall);
guard("tfefl",enall);

/* immediate transitions */
imm("tpr1f1");
priority("tpr1f1",100);
imm("tpr1f2");
priority("tpr1f2",100);

imm("tpr2f1");
priority("tpr2f1",100);
imm("tpr2f2");
priority("tpr2f2",100);

/* parameters for the transitions */
rateval("tmm1fl",1000./2400.);
rateval("tsw1fl",1000./2400.);
ratedep("tpr1fl",1000./2400.,"pr1up");

rateval("tmm1r",1000.);
rateval("tsw1r",1000.);
rateval("tpr1r",1000.);

rateval("tmm2fl",1000./2400.);
rateval("tsw2fl",1000./2400.);
ratedep("tpr2fl",1000./2400.,"pr2up");

rateval("tmm2r",1000.);
rateval("tsw2r",1000.);
```

```
    rateval("tpr2r",1000.);

    rateval("tdbfl",1000./2400.);
    rateval("tfefl",1000./2400.);

    probval("tpr1f1",coverage);
    probval("tpr1f2",1.0 – coverage);

    probval("tpr2f1",coverage);
    probval("tpr2f2",1.0 – coverage);

    /* input and output arcs */
    iarc("tmm1fl","mm1up");   oarc("tmm1fl","mm1dn");
    iarc("tsw1fl","sw1up");   oarc("tsw1fl","sw1dn");
    iarc("tpr1fl","pr1up");   oarc("tpr1fl","pr1tmp");
    iarc("tpr1f1","pr1tmp");   oarc("tpr1f1","pr1dn1");
    iarc("tpr1f2","pr1tmp");   oarc("tpr1f2","pr1dn2");
    iarc("tpr1f2","dbup");   oarc("tpr1f2","dbdn");

    iarc("tmm1r","mm1dn");   oarc("tmm1r","mm1up");
    iarc("tsw1r","sw1dn");   oarc("tsw1r","sw1up");
    iarc("tpr1r","pr1dn1");   oarc("tpr1r","pr1up");
    harc("tpr1r","mm1dn");
    harc("tpr1r","mm2dn");
    harc("tpr1r","sw1dn");
    harc("tpr1r","sw2dn");

    iarc("tmm2fl","mm2up");   oarc("tmm2fl","mm2dn");
    iarc("tsw2fl","sw2up");   oarc("tsw2fl","sw2dn");
    iarc("tpr2fl","pr2up");   oarc("tpr2fl","pr2tmp");
    iarc("tpr2f1","pr2tmp");   oarc("tpr2f1","pr2dn1");
    iarc("tpr2f2","pr2tmp");   oarc("tpr2f2","pr2dn2");
    iarc("tpr2f2","dbup");      oarc("tpr2f2","dbdn");

    iarc("tmm2r","mm2dn");   oarc("tmm2r","mm2up");
    iarc("tsw2r","sw2dn");   oarc("tsw2r","sw2up");
    iarc("tpr2r","pr2dn1");   oarc("tpr2r","pr2up");
    harc("tpr2r","mm1dn");
    harc("tpr2r","mm2dn");
    harc("tpr2r","sw1dn");
    harc("tpr2r","sw2dn");

    iarc("tdbfl","dbup");   oarc("tdbfl","dbdn");
    iarc("tfefl","feup");   oarc("tfefl","fedn");

}

int assert() {

    /* count the number of states in which the failure of the database
     by itself has caused system failure.  This excludes the states
     in which the database has been corrupted by a failing processor
     */

    if ( mark("dbdn") == 1 &&
            mark("pr1dn2") == 0 &&
            mark("pr2dn2") == 0 )
        count++;
    return(RES_NOERR);
}

void ac_init() {
    fprintf(stderr,"\nExample from Heidelberger & Goyal\n\n");
```

```
}

void ac_reach() {
}

double reliab() {

   /* if the database is failed */
   if ( mark("dbup") == 0 )
      return(0.0);

   /* if the front end is failed */
   if ( mark("feup") == 0 )
      return(0.0);

   /* if both the processing sub-systems are failed */
   if (( mark("mm1up") == 0 || mark("sw1up") == 0 ||
         mark("pr1up") == 0 ) &&
         ( mark("mm2up") == 0 || mark("sw2up") == 0 ||
         mark("pr2up") == 0 ))
      return(0.0);

   return(1.0);
}

void ac_final() {

   double time_pt;

   solve(INFINITY);


   for ( time_pt = 0.1; time_pt ≤ 1.0; time_pt+= 0.1 ) {
      solve(time_pt);
      pr_expected("Reliability:", reliab);
   }

   pr_value("No. of States in which DB caused failure", (double)count);

}
```

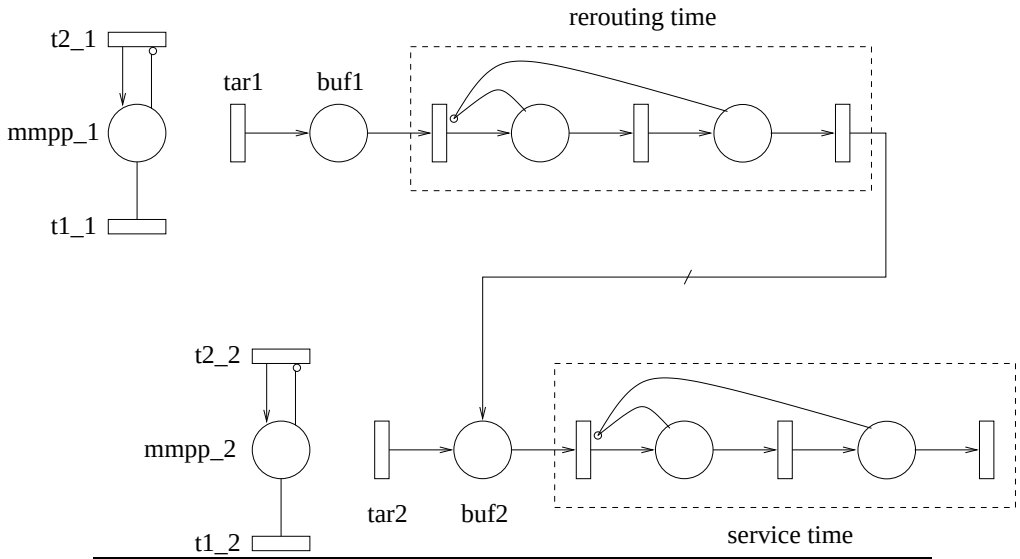
10.6 ATM network under overload

10.6.1 Source

Chang-Yu Wang, D. Logothetis, K.S. Trivedi and I. Viniotis, Transient Behavior of ATM Networks under Overloads, *Proceedings of the IEEE INFOCOM 96*, San Francisco, CA, pp. 978-985, March 1996.

10.6.2 Description

This example models ATM networks under overloads. The SPN is shown in Figure 10.9.



Transition	Rate Function	Guard Function
tar_1	if $(\#mmpp_1)$ λ_1^1 else λ_2^1	$(\#buf_1 + \sum_i \#P_{reroute[i]}) < K_1$
tar_2	if $(\#mmpp_2)$ λ_1^2 else λ_2^2	$(\#buf_2 + \sum_i \#P_{serv}[i]) < K_2$

Figure 10.9: SPN for Example 10.6

10.6.3 Features

- Transient analysis.
- Marking dependent firing rates.
- Guard function.
- General reward specification.

10.6.4 SPNP File — *atm.c*

```
#include <stdio.h>
#include "user.h"

/* global variables */

double a1=0.0269163;
```

```
double a2=0.0269163;
double b1=0.00672908;
double b2=0.00672908;
double lambda11=1.5058;
double lambda21=1.5058;
double lambda12=0.00301161;
double lambda22=0.00301161;
int r1=5;
int r2=5;
double mu1=2.73;
double mu2=2.73;
int K1=16;
int K2=16;
double e=0.0001;


/* prototype reward functions */
double Qlen1 () ;
double Earrival () ;
double Qlen2 () ;
double ELR () ;
double PFull () ;

/* prototype guard functions */
int gar2 () ;
int gar1 () ;

/* prototype rate functions */
double REr1 () ;
double Rar1 () ;
double REr2 () ;
double Rar2 () ;

/* prototype cardinality functions */
int R2 () ;
int dep12 () ;
int R1 () ;


void options() {
  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;
  /* NUMERICAL SOLUTION chosen */
  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,2000) ;
  fopt(FOP_PRECISION,0.000001) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_YES) ;
```

```
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;
}


/* REWARD Functions */
double Qlen1 ()
{
  return((double)mark("buf1") + ((double)mark("Er_token1")
              + (double)mark("Er_stage1"))/r1);
}

double Earrival ()
{
  double ret_val;
  ret_val= (mark("mmpp_2")) ? lambda21:lambda22;
  if (mark("Er_token1")==1) {
    ret_val += r1/mu1;
  }
  return(ret_val);
}

double Qlen2 ()
{
  return((double)mark("buf2") + ((double)mark("Er_token2")
              + (double)mark("Er_stage2"))/r1);
}

double ELR ()
{
  double ret_val;

  if (Qlen2()+e ≥ K2) {
    ret_val= (mark("mmpp_2")) ? lambda21:lambda22;
    if (mark("Er_token1")==1) {
      ret_val += r1/mu1;
    }
    return(ret_val);
  }
  else {
    return(0);
  }
}
double PFull ()
{
  if (Qlen2()+e≥K2) {
    return(1.0);
  }
  else {
    return(0);
  }
}


/* GUARD Functions */
int gar2 ()
```

```
{
 return(Qlen2()+e<K2);
}

int gar1 ()
{
 return(Qlen1()+e<K1);
}


/* RATE Functions */
double REr1 ()
{
 return(r1/mu1);
}

double Rar1 ()
{
 if (mark("mmpp_1")) {
  return(lambda11);
 }
 else {
  return(lambda12);
 }
}

double REr2 ()
{
 return(r2/mu2);
}

double Rar2 ()
{
 if (mark("mmpp_2")) {
  return(lambda21);
 }
 else {
  return(lambda22);
 }
}


/* CARDINALITY Functions */
int R2 ()
{
 return(r2);
}

int dep12 ()
{
 if (K2-Qlen2()+e<1) {
  return(0);
 }
 else {
  return(1);
 }
}

int R1 ()
{
```

```
 return(r1);
}

void net() {
 /* PLACE */
 place("mmpp_1");
 init("mmpp_1",1);
 place("mmpp_2");
 init("mmpp_2",1);
 place("buf1");
 place("Er_token1");
 place("Er_stage1");
 place("buf2");
 place("Er_token2");
 place("Er_stage2");
 /* TRANSITION */
 rateval("t2_1",b1);
 rateval("t2_2",b2);
 rateval("t1_1",a1);
 rateval("t1_2",a2);
 ratefun("tar1",Rar1);
 guard("tar1",gar1);
 imm("Er_in1");
 priority("Er_in1",20);
 probval("Er_in1",1.);
 ratefun("Er_trans1",REr1);
 imm("Er_out1");
 priority("Er_out1",20);
 probval("Er_out1",1.);
 ratefun("tar2",Rar2);
 guard("tar2",gar2);
 imm("Er_in2");
 priority("Er_in2",20);
 probval("Er_in2",1.);
 ratefun("Er_trans2",REr2);
 imm("Er_out2");
 priority("Er_out2",20);
 probval("Er_out2",1.);
 /* ARC */
 oarc("t2_1","mmpp_1");
 iarc("t1_1","mmpp_1");
 oarc("t2_2","mmpp_2");
 iarc("t1_2","mmpp_2");
 harc("t2_1","mmpp_1");
 harc("t2_2","mmpp_2");
 oarc("tar1","buf1");
 iarc("Er_in1","buf1");
 voarc("Er_in1","Er_token1",R1);
 iarc("Er_trans1","Er_token1");
 oarc("Er_trans1","Er_stage1");
 viarc("Er_out1","Er_stage1",R1);
 voarc("Er_out1","buf2",dep12);
 oarc("tar2","buf2");
 iarc("Er_in2","buf2");
 voarc("Er_in2","Er_token2",R2);
 iarc("Er_trans2","Er_token2");
 oarc("Er_trans2","Er_stage2");
 viarc("Er_out2","Er_stage2",R2);
 harc("Er_in1","Er_token1");
 harc("Er_in1","Er_stage1");
 harc("Er_in2","Er_token2");
 harc("Er_in2","Er_stage2");
}
```

```
int assert()
{
 return(RES_NOERR);
}

void ac_init()
{
}

void ac_reach() {

}

void ac_final() {

  float ispn_count=0.0, intvl = 10.0;

  while(ispn_count ≤ 200.0) {

    solve(ispn_count);

    pr_expected("Queue Len1", Qlen1);
    pr_expected("Queue Len2", Qlen2);

    pr_expected("ELR", ELR);
    pr_expected("PFull", PFull);
    pr_expected("Earrival", Earrival);

    ispn_count = ispn_count + intvl;
  }
}
```

10.7 Criticality Importance and Birnbaum Importance

10.7.1 Source

R. M. Fricks and K. S. Trivedi, On Computing Importance Measures Using Reward Models, *VII Simposio de Computadores Tolerantes a Falhas (VII SCTF)*, pp. 169 – 183, Campina Grande, Brazil, Jul. 1997.

10.7.2 Description

A novel technique for computing importance measures in state space dependability models is introduced here. Specifically, reward functions in a Markov reward model are utilized for this purpose, in contrast to the common method of computing importance measures

through combinatorial models and structure functions. The following simple example is used to show how to calculate Criticality Importance and Birnbaum Importance.

10.7.3 Features

- Define function with a Stochastic Petri net
- Reward based measures.

10.7.4 SPNP File — *sun.c*

```
#include <stdio.h>
#include "user.h"

/*————————————*
 * REWARD RATE FUNCTIONS *
 *————————————*/

/* Criticality */
double Q1() { return mark("p1") == 1 ? 1. : 0.; }
double Q2() { return mark("p2") == 1 ? 1. : 0.; }
double Q3() { return mark("p3") == 1 ? 1. : 0.; }
double Q()  { return Q1()+Q2()+Q3() ≥ 2. ? 1. : 0.; }

/* Birnbaum */
double g11()  { return 1.+Q2()+Q3() ≥ 2. ? 1. : 0.; }
double g10()  { return 0.+Q2()+Q3() ≥ 2. ? 1. : 0.; }
double g21()  { return Q1()+1.+Q3() ≥ 2. ? 1. : 0.; }
double g20()  { return Q1()+0.+Q3() ≥ 2. ? 1. : 0.; }
double g31()  { return Q1()+Q2()+1. ≥ 2. ? 1. : 0.; }
double g30()  { return Q1()+Q2()+0. ≥ 2. ? 1. : 0.; }

void node(failure,down,lambda)
  float lambda;
  char *failure,*down;
{
  rateval(failure,lambda);
  place(down);

  oarc(failure,down);
  harc(failure,down);
}
/*————————————*
 * SPNP FUNCTIONS *
 *————————————*/

void options() {
       iopt(IOP_PR_RGRAPH,VAL_NO);
       iopt(IOP_PR_MC,VAL_NO);
       iopt(IOP_ITERATIONS,10000);
       fopt(FOP_PRECISION,0.000000000001);
}

void net() {
```

```
  node("t1","p1",0.001);
  node("t2","p2",0.002);
  node("t3","p3",0.003);
}

int assert() {

  return(RES_NOERR);
}

void ac_init() {
}

void ac_reach() {
}

void ac_final() {
        double b1,b2,b3,q;

        solve(20.);
        b1 = expected(g11)−expected(g10);
        b2 = expected(g21)−expected(g20);
        b3 = expected(g31)−expected(g30);
        printf("Birnbaum:\n");
        printf("c1:%f\n",b1);
        printf("c2:%f\n",b2);
        printf("c3:%f\n",b3);

        q = expected(Q);
        printf("Criticality:\n");
        printf("c1%f\n",b1∗expected(Q1)/q);
        printf("c2%f\n",b2∗expected(Q2)/q);
        printf("c3%f\n",b3∗expected(Q3)/q);
}
```

10.8 Channel recovery scheme in a cellular network

10.8.1 Source

Y. Ma, C. W. Ro and K. S. Trivedi, Performability Analysis of Channel Allocation with Channel Recovery Strategy in Cellular Network, *Proceedings of IEEE 1998 International Conference on Universal Personal Communications (ICUPC'98)*, Florence, Italy, 5-9 October, 1998.

10.8.2 Description

The net is shown in Figure 10.10

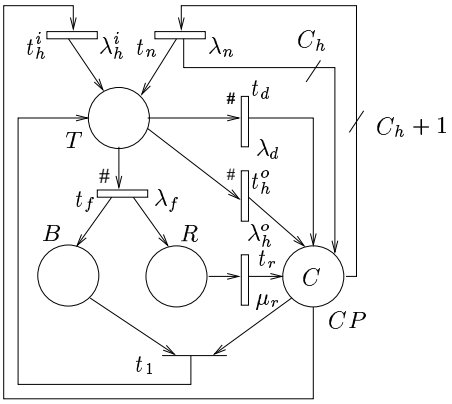


Figure 10.10: SPN for a channel recovery scheme in a cellular network.

10.8.3 Features

- Fixed point iteration. The handoff arrival rate (λ_h^i) of transition t_h^i equals to the throughput of transition t_h^o, which is used to represent the departure of handoff calls.
- Reward based functions to compute expected values.
- Default measures
- Steady-state analysis
- Use of parameters and late binding (By **parm()**, **useparm()**, and **bind()**) to reuse the SPN model.

10.8.4 SPNP File — *icupc98.c*

```
/* This is the SPNP source code for the first recovery scheme adopted from
   Ma, Ro and Trivedi "Performability Analysis of Channel Allocation with
   Channel Recovery Strategy in Cellular Networks", In Proceedings
   of IEEE 1998 International Conference on Universal Personal
   Communications (ICUPC'98), Florence, Italy, 5-9 October, 1998.
*/

#include "user.h"
#include <math.h>

#define MAX_ITERATIONS 6
#define MAX_ERROR 1e-7

/* Global variables */
double lam_n, lam_h_o, lam_d, lam_f, h_b, mu_r, lam_h_i, tmp;
int t_channel, g_c, sym;
FILE *s1_in, *s1_diff;
```

```
void options() {
  iopt(IOP_SSMETHOD, VAL_SSSOR);
  iopt(IOP_TSMETHOD,VAL_TSUNIF);
  iopt(IOP_OK_TRANS_M0, VAL_YES);
  iopt(IOP_MC,VAL_CTMC);
  iopt(IOP_ITERATIONS,20000);
  fopt(FOP_ABS_RET_M0,0.0);
  fopt(FOP_PRECISION,MAX_ERROR*1e-3);

  t_channel=28;
  g_c = 1;
  lam_n = 10;  /* New call arrival rate */

  lam_h_o = 0.33;   /* handoff every 5 minutes */

  lam_h_i = 0.2; /* Handoff_in rate */

  lam_d=0.5; /* call duration: 120 seconds */

  lam_f=0.000016677;
  mu_r = 0.0167;
}

void net() {

  /* parameters */
  parm("lam_h_i");
  parm("lam_n");

  place("T");    place("B");    place("R");
  place("CP");       init("CP",t_channel) ;

  /* timed trans */
  rateval("t_n", 1.0);
  useparm("t_n", "lam_n");
  rateval("t_h_i", 1.0);
  useparm("t_h_i","lam_h_i");
  ratedep("t_d", lam_d, "T");
  ratedep("t_f", lam_f, "T");
  ratedep("t_h_o", lam_h_o,"T");
  rateval("t_r", mu_r);

   /* immed trans */
  imm("t_1") ; priority("t_1", 100) ;
  probval("t_1",1.0);

  /*  ARC  form timed trans*/
  miarc("t_n","CP", g_c+1); oarc("t_n","T"); moarc("t_n","CP",g_c);
  iarc("t_h_i","CP"); oarc("t_h_i","T");
  iarc("t_h_o","T"); oarc("t_h_o","CP");
  iarc("t_d","T");  oarc("t_d","CP");
  iarc("t_f","T");  oarc("t_f","B");     oarc("t_f","R");
  iarc("t_r","R");  oarc("t_r","CP");

  /*  ARC  for immediate trans */
  iarc("t_1","B");  iarc("t_1", "CP"); oarc("t_1", "T");

    /* assign parameters */
  bind("lam_h_i", lam_h_i);
  bind("lam_n", lam_n);

  }
```

```
int assert() {
  return(RES_NOERR);
}

void ac_init() {
}

void ac_reach() {
}

double BH()
{
  if (mark("CP")==0)
    return (1.0);
  else
    return (0.0);
}

double BN()
{
  if (mark("CP")≤g_c)
    return (1.0);
  else
    return (0.0);
}

double ACh()
{
  return(mark("CP"));
}

double hotput()
{
  return(rate("t_h_o"));
}

/* average failure arrival rate */
double ftput()
{
  return(rate("t_f"));
}

double fnum()
{
  return(mark("B"));
}

void ac_final() {
  int i;
  double tp,err;

  for (i=1; i<MAX_ITERATIONS; i++) {
    pr_value("lam_h_i", lam_h_i);
    bind("lam_h_i", lam_h_i);
    solve(INFINITY);
    tp = expected(hotput);
    pr_value("Throughput of t_h_o", tp);
    err = fabs((lam_h_i−tp)/tp);
    pr_value("Error", err);
    if( err < MAX_ERROR ) break;
```

```
    lam_h_i = tp;
  }

  pr_expected("block handoff:", BH);
  pr_expected("block new:", BN);
  pr_expected("available channel: ", ACh);
  pr_value("avg. waiting time:", expected(fnum)/expected(ftput));
}
```

10.9 Accurate Model for the BUS in ATM LAN emulation

10.9.1 Source

H. Sun, X. Zang. and K.S. Trivedi. "Performance of Broadcast and Unknown Server (BUS) in ATM LAN Emulation", Technical Report. *Center for Advanced Computing and Communication, Duke University*, 1999.

10.9.2 Description

The net shown in Figure 10.11 represents a performance model of the Broadcast and Unknown Server (BUS) in the ATM LAN emulator.

10.9.3 Features

- Markovian model
- Dependent-marking rates
- Priorities

10.9.4 SPNP File — *LAN.c*

```
#include <stdio.h>
#include "user.h"

/* corresponds to ACCURATE model in the paper*/

#define PLST 2 /* Short Packet Size */
#define PLMD 4 /* Medium Packet Size */
#define PLLG 32 /* Long Packet Size */

#define c1 0.342
#define c2 0.093
#define c3 0.565
```

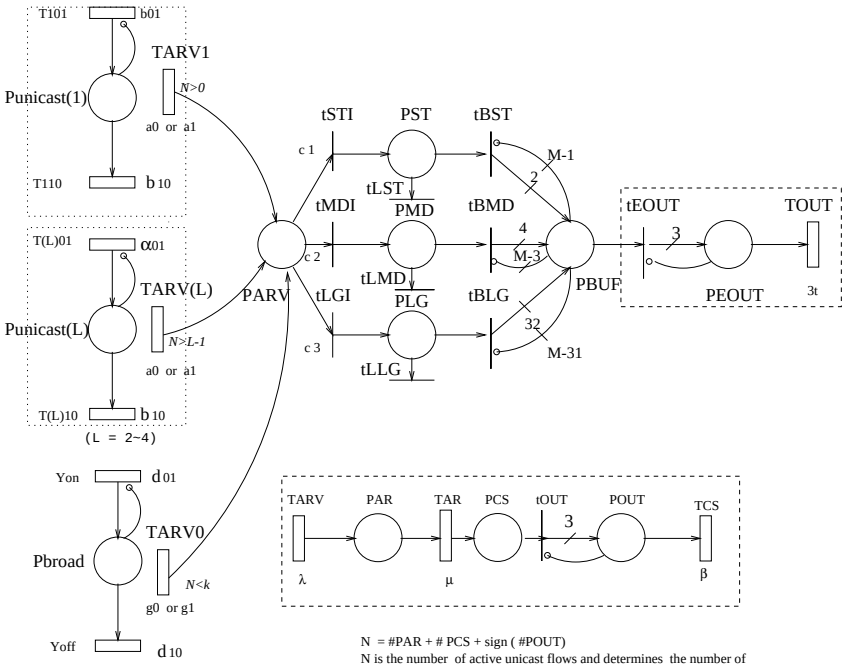


Figure 10.11: Accurate SRN Model for the BUS

```
#define rateOUT 5000

int K=2;
double lambda=0.9, mu=1, beta=1.1;

int BN=40; /* Buffer Size */
int QLEN;

double g0=0.01, g1=0.05, d01=0.5, d10=0.5;
double a0=0.01, a1=0.05, b01=0.5, b10=0.5;

 /* prototype guard functions */
 int guardARV ();

 int  guardARV1 () ;
 int  guardARV2 () ;
 int  guardARV3 () ;
 int  guardARV4 () ;

 int guardBST () ;
 int guardBMD () ;
 int guardBLG () ;

 /* prototype rate functions */
 double rateARV0 () ;
 double rateARV1 () ;
 double rateARV2 () ;
 double rateARV3 () ;
 double rateARV4 () ;
```

```
void options() {

  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;

  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,20000) ;
  fopt(FOP_PRECISION,1e-13) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_YES) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;

}


/* REWARD Functions */
double QL () {
   if (mark("PBUF") > QLEN)
      return(1.0);
   else
      return(0);
}
double MLP () {
   if (mark("PBUF") > BN - PLMD)
      return(1.0);
   else
      return(0);
}
double SLP () {
   if (mark("PBUF") > BN - PLST)
      return(1.0);
   else
      return(0);
}
double LLP () {
   if (mark("PBUF") > BN - PLLG)
      return(1.0);
   else
      return(0);
}

/* GUARD Functions */
```

```
int guardARV ()
{
 int n = 0;
 if (mark("POUT")>0)
   n = 1;
 if (mark("PAR") + mark("PCS") + n < K)
   return(1);
 else
   return(0);
}

int guardARV1 () {
int n = 0;
 if (mark("POUT")>0) n = 1;
 n+=mark("PAR") + mark("PCS");
   if (n>0)
      return(1);
   else
      return(0);
}

int guardARV2 () {
int n = 0;
 if (mark("POUT")>0) n = 1;
 n+=mark("PAR") + mark("PCS");
   if (n>1)
      return(1);
   else
      return(0);
}

int guardARV3 () {
int n = 0;
 if (mark("POUT")>0) n = 1;
 n+=mark("PAR") + mark("PCS");
   if (n>2)
      return(1);
   else
      return(0);
}

int guardARV4 () {
int n = 0;
 if (mark("POUT")>0) n = 1;
 n+=mark("PAR") + mark("PCS");
   if (n>3)
      return(1);
   else
      return(0);
}

int guardBST () {
   if (mark("PBUF") ≤ BN − PLST)
      return(1);
   else
      return(0);
}
int guardBMD () {
   if (mark("PBUF") ≤ BN − PLMD)
      return(1);
   else
      return(0);
```

```
}
int guardBLG () {
   if (mark("PBUF") ≤ BN − PLLG)
      return(1);
   else
      return(0);
}

/* RATE Functions */



double rateARV0 () {

  if (mark("PON0"))
      return(g1);
  else
      return(g0);

}

double rateARV1 () {

  if (mark("PON1"))
      return(a1);
  else
      return(a0);

}
double rateARV2 () {

  if (mark("PON2"))
      return(a1);
  else
      return(a0);

}
double rateARV3 () {

  if (mark("PON3"))
      return(a1);
  else
      return(a0);

}

double rateARV4 () {

  if (mark("PON4"))
      return(a1);
  else
      return(a0);

}

void net() {

  /* PLACE */
 place("PAR") ;
 place("PCS") ;
 place("POUT") ;
```

```
place("PARV");
place("PST");
place("PMD");
place("PLG");
place("PBUF");
place("PEOUT");

place("PON0");
place("PON1");
place("PON2");
place("PON3");
place("PON4");

/* TRANSITION */
rateval("TARV",lambda);
guard("TARV", guardARV);
rateval("TAR",mu);
rateval("TCS",beta);
imm("tOUT");
priority("tOUT",20);
probval("tOUT",1.);

ratefun("TARV0",rateARV0);

ratefun("TARV1",rateARV1);
guard("TARV1",guardARV1);

ratefun("TARV2",rateARV2);
guard("TARV2",guardARV2);

ratefun("TARV3",rateARV3);
guard("TARV3",guardARV3);

ratefun("TARV4",rateARV4);
guard("TARV4",guardARV4);

rateval("T001",d01);
rateval("T010",d10);

rateval("T101",b01);
rateval("T110",b10);

rateval("T201",b01);
rateval("T210",b10);

rateval("T301",b01);
rateval("T310",b10);

rateval("T401",b01);
rateval("T410",b10);


imm("tST");
priority("tST",20);
probval("tST",c1);
imm("tMD");
priority("tMD",20);
probval("tMD",c2);
imm("tLG");
priority("tLG",20);
probval("tLG",c3);
imm("tBST");
priority("tBST",40);
probval("tBST",1.);
```

```
guard("tBST",guardBST);
imm("tBMD");
priority("tBMD",40);
probval("tBMD",1.);
guard("tBMD",guardBMD);
imm("tBLG");
priority("tBLG",40);
probval("tBLG",1.);
guard("tBLG",guardBLG);
imm("tLST");
priority("tLST",20);
probval("tLST",1.);
imm("tLMD");
priority("tLMD",20);
probval("tLMD",1.);
imm("tLLG");
priority("tLLG",20);
probval("tLLG",1.);
imm("tEOUT");
priority("tEOUT",20);
probval("tEOUT",1.);
rateval("TOUT",rateOUT);


/* ARC */

oarc("TARV","PAR");
iarc("TAR","PAR");
oarc("TAR","PCS");
iarc("tOUT","PCS");
moarc("tOUT","POUT", 3);
iarc("TCS","POUT");
harc("tOUT","POUT");

oarc("T001","PON0");
iarc("T010","PON0");
harc("T001","PON0");

oarc("T101","PON1");
iarc("T110","PON1");
harc("T101","PON1");

oarc("T201","PON2");
iarc("T210","PON2");
harc("T201","PON2");

oarc("T301","PON3");
iarc("T310","PON3");
harc("T301","PON3");

oarc("T401","PON4");
iarc("T410","PON4");
harc("T401","PON4");

oarc("TARV0","PARV");
oarc("TARV1","PARV");
oarc("TARV2","PARV");
oarc("TARV3","PARV");
oarc("TARV4","PARV");

iarc("tST","PARV");
iarc("tMD","PARV");
iarc("tLG","PARV");
oarc("tST","PST");
```

```
  oarc("tMD","PMD");
  oarc("tLG","PLG");
  iarc("tBST","PST");
  iarc("tLST","PST");
  iarc("tBMD","PMD");
  iarc("tLMD","PMD");
  iarc("tBLG","PLG");
  iarc("tLLG","PLG");
  moarc("tBST","PBUF",PLST);
  moarc("tBLG","PBUF",PLLG);
  moarc("tBMD","PBUF",PLMD);
  iarc("tEOUT","PBUF");
  moarc("tEOUT","PEOUT",3);
  iarc("TOUT","PEOUT");
  harc("tEOUT","PEOUT");

}

int assert() {
  return(RES_NOERR);
}

void ac_init() {

}

void ac_reach() {

}


void ac_final() {
  double llp, mlp, slp, pql;
  int dBN = BN/20;

  if (dBN<1) dBN=1;

  solve(INFINITY);
  llp = expected(LLP);
  mlp = expected(MLP);
  slp = expected(SLP);

  pr_value("Loss Prob. of Short Packets (S)", slp);
  pr_value("Loss Prob. of Medium Packets (M)", mlp);
  pr_value("Loss Prob. of Long Packets (L)", llp);
  pr_value("Loss Prob. of total Packets (T)", c1 * slp + c2 * mlp + c3 * llp);

  for (QLEN = 0; QLEN < BN; QLEN += dBN)
    {
      pql = expected(QL);
      fprintf(Outfile,"\nProb. for Queue Len (>%d) =%.12g\n", QLEN, pql);
    }

  QLEN = BN - 1;
  pql = expected(QL);
  fprintf(Outfile,"\nProb. for Queue Len (=%d) =%.12g\n", BN, pql);

  pr_std_average();
}
```

10.10 Birth-death Model for the BUS in ATM LAN emulation

10.10.1 Source

H. Sun, X. Zang. and K.S. Trivedi. "Performance of Broadcast and Unknown Server (BUS) in ATM LAN Emulation", Technical Report. *Center for Advanced Computing and Communication, Duke University*, 1999.

10.10.2 Description

The net shown in Figure 10.12 represents a performance model of the Broadcast and Unknown Server (BUS) in the ATM LAN emulator. It is an simplification of Example 10.9 where the aggregation of the flows of the unicast traffic is approximated by a birth and death process.

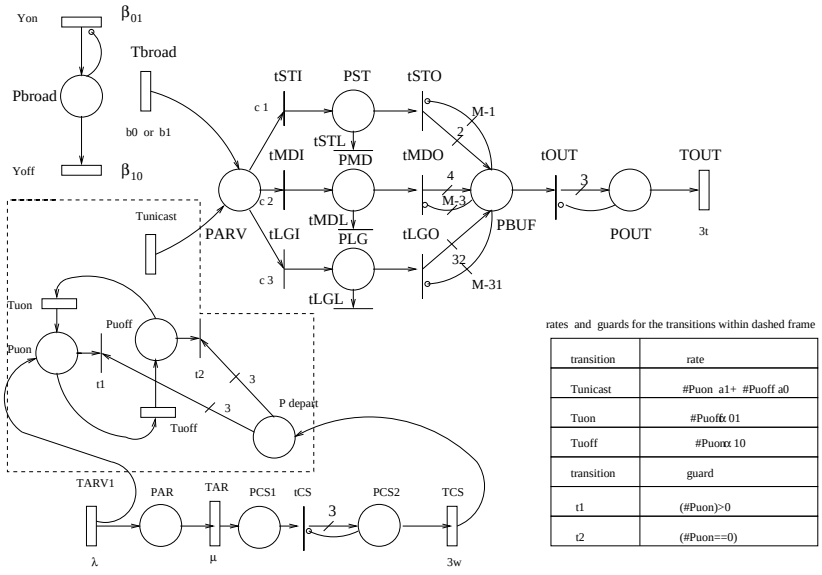


Figure 10.12: Birth-death Model for the BUS in ATM LAN emulation

10.10.3 Features

- Markovian model

- Dependent-marking rates
- Priorities

10.10.4 SPNP File — *LA.c*

```
#include <stdio.h>
#include "user.h"

/* corresponds to birth-death model in the paper*/

#define PLST 2 /* Short Packet Size */
#define PLMD 4 /* Medium Packet Size */
#define PLLG 32 /* Long Packet Size */

#define c1 0.342
#define c2 0.093
#define c3 0.565

#define rateOUT 5000

int K=3;
double lambda=0.9, mu=1, beta=1.1;

int BN=40; /* Buffer Size */
int QLEN;

double g0=0.01, g1=0.05, beta_01=0.5, beta_10=0.5;
double a0=0.01, a1=0.05, alpha_01=0.5, alpha_10=0.5;

  /* prototype guard functions */
  int guardARV ();

  int  guardt1() ;
  int  guardt2 () ;
  int  guardTunicast () ;
  int guardBST () ;
  int guardBMD () ;
  int guardBLG () ;

  /* prototype rate functions */
  double rateARV0 () ;
  double rateTunicast () ;


void options() {

  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;
```

```
  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,20000) ;
  fopt(FOP_PRECISION,1e−13) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_YES) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;



}


/* REWARD Functions */
double QL () {
   if (mark("PBUF") > QLEN)
      return(1.0);
   else
      return(0);
}
double MLP () {
   if (mark("PBUF") > BN − PLMD)
      return(1.0);
   else
      return(0);
}
double SLP () {
   if (mark("PBUF") > BN − PLST)
      return(1.0);
   else
      return(0);
}
double LLP () {
   if (mark("PBUF") > BN − PLLG)
      return(1.0);
   else
      return(0);
}

/* GUARD Functions */
int guardARV ()
{
 int n = 0;
 if (mark("PCS2")>0)
   n = 1;
 if (mark("PAR") + mark("PCS1") + n < K)
   return(1);
 else
   return(0);
}

int guardt2 () {
 if (mark("Puon")==0)
      return(1);
```

```
    else
        return(0);
}
int guardTunicast () {
    if (mark("Puon")>0)
            return(1);
    else
        return(0);
}

int guardSTO () {
    if (mark("PBUF") ≤ BN − PLST)
        return(1);
    else
        return(0);
}
int guardMDO () {
    if (mark("PBUF") ≤ BN − PLMD)
        return(1);
    else
        return(0);
}
int guardLGO () {
    if (mark("PBUF") ≤ BN − PLLG)
        return(1);
    else
        return(0);
}

/* RATE Functions */



double rateTbroad () {

  if (mark("Pbroad"))
        return(g1);
  else
        return(g0);

}

double rateTunicast () {
double x= mark("Puon")*a1+mark("Puoff")*a0;
return (x);
}


void net() {

   /* PLACE */
  place("PAR") ;
  place("PCS1") ;
  place("PCS2") ;

  place("PARV") ;
  place("PST") ;
  place("PMD") ;
  place("PLG") ;
  place("PBUF") ;
  place("POUT") ;
```

```
place("Pbroad");

place("Puon");
place("Puoff");
place("Pdepart");

/* TRANSITION */
rateval("TARV1",lambda);
guard("TARV1", guardARV);
rateval("TAR",mu);
rateval("TCS",beta);
imm("tCS");
priority("tCS",20);
probval("tCS",1.);

ratefun("Tbroad",rateTbroad);

rateval("Yon",beta_01);
rateval("Yoff",beta_10);

ratedep("Tuon",alpha_01,"Puoff");

ratedep("Tuoff",alpha_10,"Puon");

ratefun("Tunicast",rateTunicast);
guard("Tunicast",guardTunicast);

imm("t1");
probval("t1",1.);
priority("t1",20);

imm("t2");
probval("t2",1.);
priority("t2",20);
guard("t2",guardt2);


imm("tSTI");
priority("tSTI",20);
probval("tSTI",c1);
imm("tMDI");
priority("tMDI",20);
probval("tMDI",c2);
imm("tLGI");
priority("tLGI",20);
probval("tLGI",c3);
imm("tSTO");
priority("tSTO",40);
probval("tSTO",1.);
guard("tSTO",guardSTO);
imm("tMDO");
priority("tMDO",40);
probval("tMDO",1.);
guard("tMDO",guardMDO);
imm("tLGO");
priority("tLGO",40);
probval("tLGO",1.);
guard("tLGO",guardLGO);
imm("tSTL");
priority("tSTL",20);
probval("tSTL",1.);
imm("tMDL");
priority("tMDL",20);
probval("tMDL",1.);
```

```
 imm("tLGL");
 priority("tLGL",20);
 probval("tLGL",1.);
 imm("tOUT");
 priority("tOUT",20);
 probval("tOUT",1.);
 rateval("TOUT",rateOUT);


 /* ARC */

 oarc("TARV1","PAR");
 iarc("TAR","PAR");
 oarc("TAR","PCS1");
 iarc("tCS","PCS1");
 moarc("tCS","PCS2", 3);
 iarc("TCS","PCS2");
 harc("tCS","PCS2");

 oarc("Yon","Pbroad");
 iarc("Yoff","Pbroad");
 harc("Yon","Pbroad");


 oarc("Tbroad","PARV");

 oarc("TARV1","Puon");
 oarc("TCS","Pdepart");
 oarc("Tuon","Puon");
 iarc("Tuoff","Puon");
 iarc("t1","Puon");
 miarc("t1","Pdepart",3);
 iarc("t2","Puoff");
 miarc("t2","Pdepart",3);
 oarc("Tuoff","Puoff");
 iarc("Tuon","Puoff");

 oarc("Tunicast","PARV");


 iarc("tSTI","PARV");
 iarc("tMDI","PARV");
 iarc("tLGI","PARV");
 oarc("tSTI","PST");
 oarc("tMDI","PMD");
 oarc("tLGI","PLG");
 iarc("tSTO","PST");
 iarc("tSTL","PST");
 iarc("tMDO","PMD");
 iarc("tMDL","PMD");
 iarc("tLGO","PLG");
 iarc("tLGL","PLG");
 moarc("tSTO","PBUF",PLST);
 moarc("tLGO","PBUF",PLLG);
 moarc("tMDO","PBUF",PLMD);
 iarc("tOUT","PBUF");
 moarc("tOUT","POUT",3);
 iarc("TOUT","POUT");
 harc("tOUT","POUT");

}

int assert() {
```

```
  return(RES_NOERR);
}

void ac_init() {

}

void ac_reach() {

}


void ac_final() {
  double llp, mlp, slp, pql;

  solve(INFINITY);
  llp = expected(LLP);
  mlp = expected(MLP);
  slp = expected(SLP);

  pr_value("Loss Prob. of Short Packets (S)", slp);
  pr_value("Loss Prob. of Medium Packets (M)", mlp);
  pr_value("Loss Prob. of Long Packets (L)", llp);
  pr_value("Loss Prob. of total Packets (T)", c1 * slp + c2 * mlp + c3 * llp);

  for (QLEN = 0; QLEN < BN; QLEN += 5)
    {
      pql = expected(QL);
      fprintf(Outfile,"\nProb. for Queue Len (> %d) = %.12g\n", QLEN, pql);
    }

  QLEN = BN - 1;
  pql = expected(QL);
  fprintf(Outfile,"\nProb. for Queue Len (= %d) = %.12g\n", BN, pql);

  pr_std_average();
}
```

10.11 MMPP Model for the BUS in ATM LAN emulation

10.11.1 Source

H. Sun, X. Zang. and K.S. Trivedi. "Performance of Broadcast and Unknown Server (BUS) in ATM LAN Emulation", Technical Report. *Center for Advanced Computing and Communication, Duke University*, 1999.

10.11.2 Description

The net shown in Figure 10.13 represents a performance model of the Broadcast and Unknown Server (BUS) in the ATM LAN emulator. It is an simplification of Example 10.9

where the superpositiom of multiple MMPPs is approximated by a two-state MMPP.

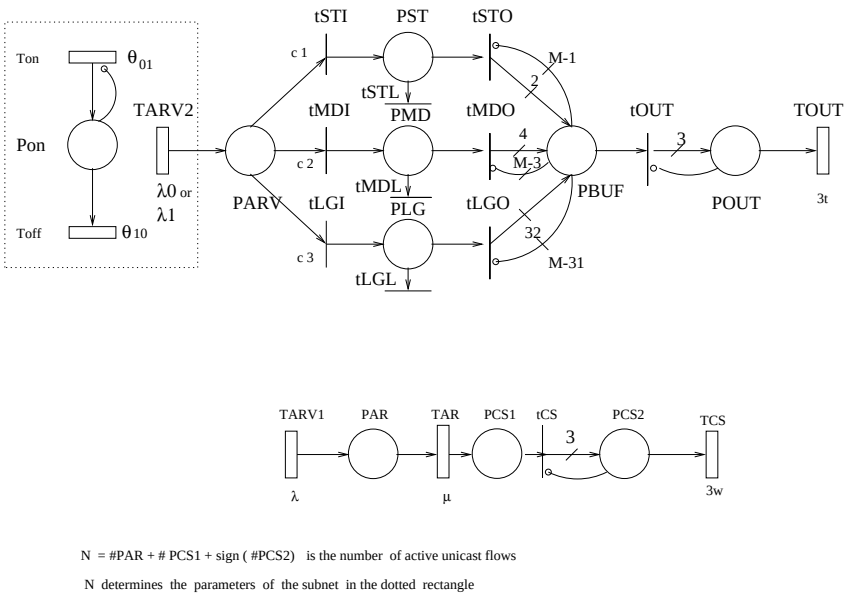


Figure 10.13: MMPP Model for the BUS in ATM LAN emulation

10.11.3 Features

- Markovian model
- Dependent-marking rates
- Priorities

10.11.4 SPNP File — *LANE.c*

```
#include <stdio.h>
#include "user.h"

/* corresponds to MMPP model in paper*/

#define PLST 2 /* Short Packet Size */
#define PLMD 4 /* Medium Packet Size */
#define PLLG 32 /* Long Packet Size */

#define c1 0.342
#define c2 0.093
#define c3 0.565

#define rateOUT 5000
```

```
int K=3;
double lambda=0.9, mu=1, beta=1.1;

int BN=100; /* Buffer Size */
int QLEN;

double g0=0.01, g1=0.05, d01=0.5, d10=0.5;
double a0=0.01, a1=0.05, b01=0.5, b10=0.5;
double m2, v2, u2, t2;
double m1, v1, u1, t1;
double rateAH, rateAL,theta0,theta1;


  /* prototype guard functions */
  int guardARV ();
  int guardARV2 () ;
  int guardBST () ;
  int guardBMD () ;
  int guardBLG () ;

  /* prototype rate functions */
  double rateARV2 () ;
  double t01 ();
  double t10 ();



void options() {

  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;

  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,20000) ;
  fopt(FOP_PRECISION,1e-13) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_YES) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;


  m1 = (b01 * a1 + b10 * a0) / (b01 + b10);
  v1 = (a1 - a0) * (a1 - a0) * b01 * b10 / ((b01 + b10) * (b01 + b10));
  u1 = (a1 * a1 * a1 * b01 + a0 * a0 * a0 * b10) / (b01 + b10);
  u1 = u1 - 3 * m1 * v1 - m1 * m1 * m1;
```

```
  t1 = 1 / (b01 + b10);

  m2 = (d01 * g1 + d10 * g0) / (d01 + d10);
  v2 = (g1 - g0) * (g1 - g0) * d01 * d10 / ((d01 + d10) * (d01 + d10));
  u2 = (g1 * g1 * g1 * d01 + g0 * g0 * g0 * d10) / (d01 + d10);
  u2 = u2 - 3 * m2 * v2 - m2 * m2 *m2;
  t2 = 1 / (d01 + d10);

}


/* REWARD Functions */
double QL () {
   if (mark("PBUF") > QLEN)
      return(1.0);
   else
      return(0);
}
double MLP () {
   if ((mark("PBUF") > BN - PLMD) && enabled("TARV2"))
      return(1.0);
   else
      return(0);
}
double SLP () {
   if ((mark("PBUF") > BN - PLST) && enabled("TARV2"))
      return(1.0);
   else
      return(0);
}
double LLP () {
   if ((mark("PBUF") > BN - PLLG) && enabled("TARV2"))
      return(1.0);
   else
      return(0);
}

/* GUARD Functions */
int guardARV ()
{
  int n = 0;
  if (mark("POUT")>0)
    n = 1;
  if (mark("PAR") + mark("PCS") + n < K)
    return(1);
  else
    return(0);
}

int guardARV2 () {
   if ((rateAL ≠ 0.0) || mark("PAON"))
      return(1);
   else
      return(0);
}
int guardBST () {
   if (mark("PBUF") ≤ BN - PLST)
      return(1);
   else
      return(0);
}
```

```
int guardBMD () {
   if (mark("PBUF") ≤ BN − PLMD)
      return(1);
   else
      return(0);
}
int guardBLG () {
   if (mark("PBUF") ≤ BN − PLLG)
      return(1);
   else
      return(0);
}

/∗ RATE Functions ∗/

double rateARV2 () {

double m,v,u,t,x,e;

int S = 0;

 if (mark("POUT"))
      S=1;

 S= S + mark("PAR") + mark("PCS");
 m = S ∗ m1 + m2;
 v = S ∗ v1+v2;
 u = S ∗ u1 + u2;
 t = (S ∗ v1∗t1+v2 ∗t2) /v;
 x =  u / sqrt(v ∗ v ∗ v);
 e = (x ∗ x + 2 − x ∗ sqrt(4 + x ∗ x)) / 2;
 rateAH = m + sqrt(v / e);
 rateAL = m − sqrt(v ∗ e);
 if (mark("PAON"))
      return(rateAH);
 else
      return(rateAL);

}

double t01 () {

double m,v,u,t,x,e;

int S = 0;

 if (mark("POUT"))
      S=1;

 S= S + mark("PAR") + mark("PCS");
 m = S ∗ m1 + m2;
 v = S ∗ v1+v2;
 u = S ∗ u1 + u2;
 t = (S ∗ v1 ∗ t1+v2 ∗t2) /v;
 x =  u / sqrt(v ∗ v ∗ v);
 e = (x ∗ x + 2 − x ∗ sqrt(4 + x ∗ x)) / 2;
 theta0 = 1 / (t ∗ (1 + e));
 return(theta0);

}

double t10() {
```

```
double m,v,u,t,x,e;

  int S = 0;

  if (mark("POUT"))
       S=1;

  S= S + mark("PAR") + mark("PCS");
  m = S * m1 + m2;
  v = S * v1+v2;
  u = S * u1 + u2;
  t = (S * v1*t1+v2 *t2) /v;
  x =  u / sqrt(v * v * v);
  e = (x * x + 2 - x * sqrt(4 + x * x)) / 2;
  theta1 = e / (t * (1 + e));
  return(theta1);
}

void net() {

   /*  PLACE  */
  place("PAR") ;
  place("PCS") ;
  place("POUT") ;

  place("PARV") ;
  place("PST") ;
  place("PMD") ;
  place("PLG") ;
  place("PBUF") ;
  place("PEOUT") ;
  place("PAON") ;

  /*  TRANSITION  */
  rateval("TARV",lambda) ;
  guard("TARV", guardARV);
  rateval("TAR",mu) ;
  rateval("TCS",beta) ;
  imm("tOUT") ;
  priority("tOUT",20) ;
  probval("tOUT",1.) ;

  ratefun("TARV2",rateARV2) ;
  guard("TARV2",guardARV2) ;
  imm("tST") ;
  priority("tST",20) ;
  probval("tST",c1) ;
  imm("tMD") ;
  priority("tMD",20) ;
  probval("tMD",c2) ;
  imm("tLG") ;
  priority("tLG",20) ;
  probval("tLG",c3) ;
  imm("tBST") ;
  priority("tBST",40) ;
  probval("tBST",1.) ;
  guard("tBST",guardBST) ;
  imm("tBMD") ;
  priority("tBMD",40) ;
  probval("tBMD",1.) ;
  guard("tBMD",guardBMD) ;
  imm("tBLG") ;
  priority("tBLG",40) ;
```

```
  probval("tBLG",1.);
  guard("tBLG",guardBLG);
  imm("tLST");
  priority("tLST",20);
  probval("tLST",1.);
  imm("tLMD");
  priority("tLMD",20);
  probval("tLMD",1.);
  imm("tLLG");
  priority("tLLG",20);
  probval("tLLG",1.);
  imm("tEOUT");
  priority("tEOUT",20);
  probval("tEOUT",1.);
  rateval("TOUT",rateOUT);
  ratefun("TT01",t01);
  ratefun("TT10",t10);

  /* ARC */
  oarc("TARV","PAR");
  iarc("TAR","PAR");
  oarc("TAR","PCS");
  iarc("tOUT","PCS");
  moarc("tOUT","POUT", 3);
  iarc("TCS","POUT");
  harc("tOUT","POUT");

  oarc("TARV2","PARV");
  iarc("tST","PARV");
  iarc("tMD","PARV");
  iarc("tLG","PARV");
  oarc("tST","PST");
  oarc("tMD","PMD");
  oarc("tLG","PLG");
  iarc("tBST","PST");
  iarc("tLST","PST");
  iarc("tBMD","PMD");
  iarc("tLMD","PMD");
  iarc("tBLG","PLG");
  iarc("tLLG","PLG");
  moarc("tBST","PBUF",PLST);
  moarc("tBLG","PBUF",PLLG);
  moarc("tBMD","PBUF",PLMD);
  iarc("tEOUT","PBUF");
  moarc("tEOUT","PEOUT",3);
  iarc("TOUT","PEOUT");
  harc("tEOUT","PEOUT");
  oarc("TT01","PAON");
  iarc("TT10","PAON");
  harc("TT01","PAON");
}

int assert() {
  return(RES_NOERR);
}

void ac_init() {

}

void ac_reach() {

}
```

```
void ac_final() {
  double llp, mlp, slp, pql;

  solve(INFINITY);
  llp = expected(LLP);
  mlp = expected(MLP);
  slp = expected(SLP);

  pr_value("Loss Prob. of Short Packets (S)", slp);
  pr_value("Loss Prob. of Medium Packets (M)", mlp);
  pr_value("Loss Prob. of Long Packets (L)", llp);
  pr_value("Loss Prob. of total Packets (T)", c1 * slp + c2 * mlp + c3 * llp);

  for (QLEN = 0; QLEN < BN; QLEN += BN / 20)
    {
      pql = expected(QL);
      fprintf(Outfile,"\nProb. for Queue Len (> %d) = %.12g\n", QLEN, pql);
    }

  QLEN = BN - 1;
  pql = expected(QL);
  fprintf(Outfile,"\nProb. for Queue Len (= %d) = %.12g\n", BN, pql);

  pr_std_average();
}
```

10.12 Performance anlysis of Multi-Protocol Label Switching Network

10.12.1 Source

X. Zang and K. S. Trivedi, *The Label Allocation, Table Recovery and Differentiated Service in Multi-Protocol Label Switching Network*, Internal technical report, Duke University.

10.12.2 Description

In this example system, n input links and one output link are considered (see Fig. 10.14). The traffic on each input link is bursty. There are two types of traffic on each input link, i.e., higher-priority VPN traffic and lower-priority best-effort traffic. All these incoming traffics will be switched to one output link. There are $n + 1$ queues at the output link, i.e., n for the higher-priority VPN traffic flows and 1 shared by the lower-priority best-effort traffic flows. Each input link uses a leaky bucket to regulate the higher-priority VPN traffic. The excess VPN traffic can be merged into the shared best-effort queue. On the other hand, the excess capacity of VPN can also be used by best-effort traffic on the same input link.

Round-robin (or weighted round-robin (WRR)) policy is used to serve the higher-priority VPN traffic.

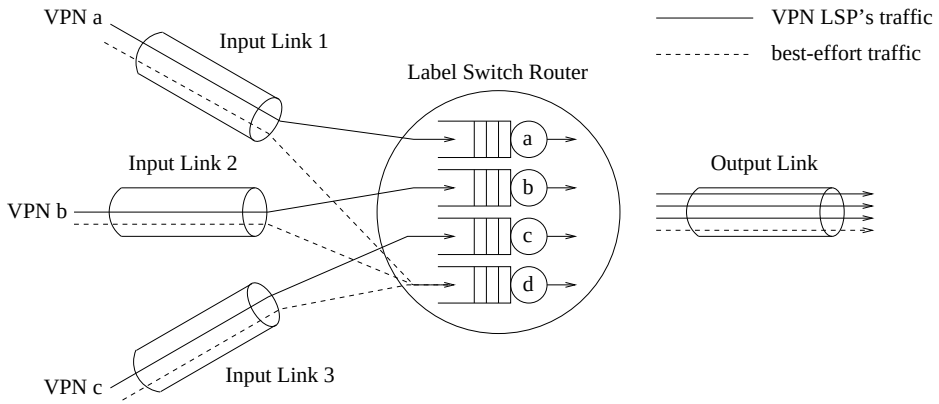


Figure 10.14: Example system

10.12.3 Features

- Utilizing model decomposition.
- Fixed point iteration.
- Script file for execution.

10.12.4 SPN model for LSN

A model directly modeling LSN will be too large to be solved. To simplify the analysis, we assume that the percentage of the higher-priority traffic is identical for all input links and the packet size distribution is equal for the traffic on all links. Then the model can be decomposed into three parts: tagged link, aggregated link and best-effort queue.

Table 10.1 gives the guards for transitions in the SPN models in Fig. 10.15, Fig. 10.16 and Fig. 10.17.

10.12.5 Files list

1. Perl script file for fixed point iteration.

```
#!/bin/env perl
$NIT = 200;
$ModelName = "smod3t";
$SPN = $ModelName . ".spn";
```

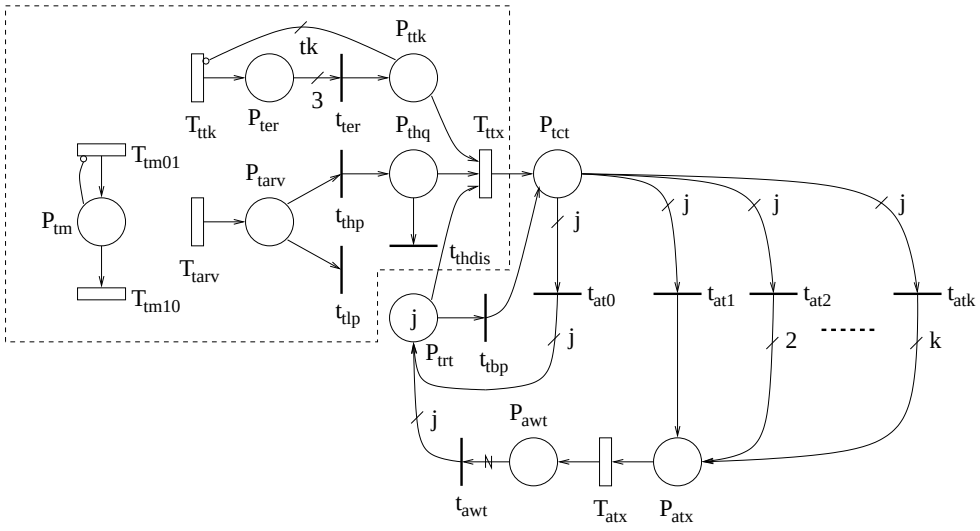


Figure 10.15: Decomposed SPN model for a tagged link

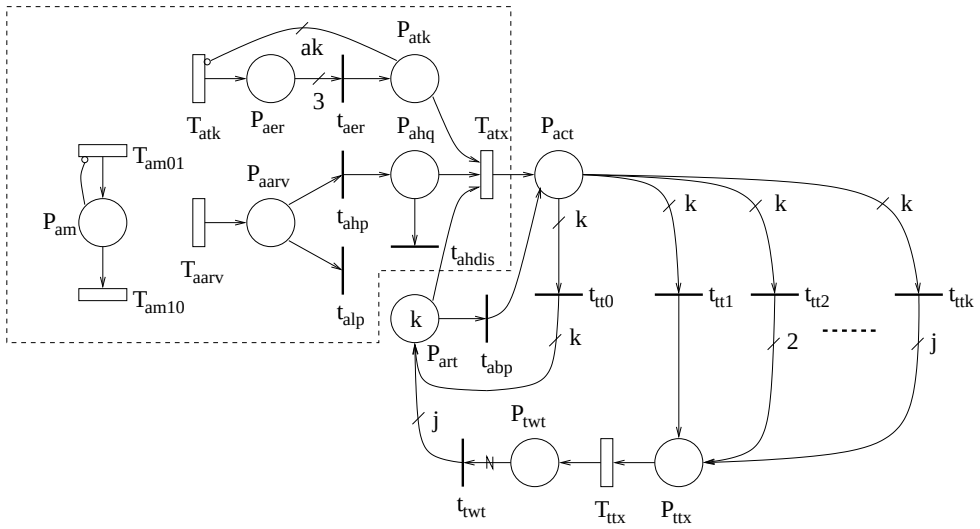


Figure 10.16: Decomposed SPN model for the aggregated links

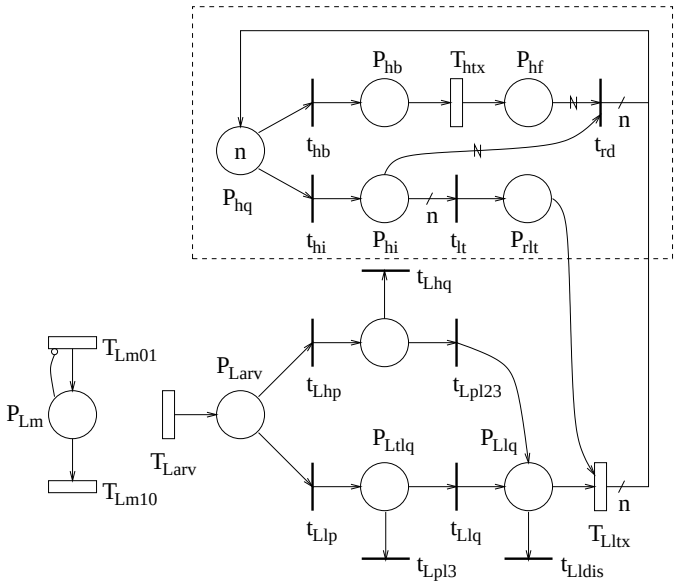


Figure 10.17: Decomposed SPN model for the best-effort queue

Transition	Guard(Policy I)	Guard (Policy II)	Guard (Policy III)
t_{thp}	1	$\#P_{thq} < B_{thq}$	$\#P_{thq} < B_{thq}$
t_{tlp}	1	1	$\#P_{thq} > \#P_{ttk}$
t_{twt}	$\#P_{atx} = 0$	as Policy I	as Policy I
t_{thdis}	$\#P_{thq} > B_{thq}$	0	0
t_{tbp}	$\#P_{thq} = 0$ or $\#P_{ttk} = 0$	as Policy I	as Policy I
t_{ahp}	1	$\#P_{ahq} < B_{ahq}$	$\#P_{ahq} < B_{ahq}$
t_{alp}	1	1	$\#P_{ahq} > \#P_{atk}$
t_{awt}	$\#P_{ttx} = 0$	as Policy I	as Policy I
t_{ahdis}	$\#P_{ahq} > B_{ahq}$	0	0
t_{abp}	$\#P_{ahq} = 0$ or $\#P_{atk} = 0$	as Policy I	as Policy I
t_{Lpl23}	0	1	1
t_{Lpl3}	0	0	1
t_{Lldis}	$\#P_{Llq} > B_{Llq}$	as Policy I	as Policy I

Table 10.1: Guards of transitions for the SPN in Fig. 10.15, Fig. 10.16 and Fig. 10.17

```
$Tagged = $ModelName . "t";
$Aggregated = $ModelName . "a";

for ($i = 0; $i ≤ $NIT; $i++)
{
  print "Fix Point Iteration: $i for Tagged Links.\n";
  system("$SPN $Tagged") == 0
    or die "finish/error exec $Tagged: $?";

  print "Fix Point Iteration: $i for Aggregated Links.\n";
  system("$SPN $Aggregated") == 0
  or die "error exec $Aggregated: $?";
}
```

2. SPNP file *smod3t.c*

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "user.h"
#include "misc.h"

/* global variables */

int NBUF = 32;

int NTK = 8;

int NRR = 1;

int NMRR = 2;

int NER = 3;

double larvrate = 3;
double harvrate = 6;

double rateTMMPP01 = 0.01;
double rateTMMPP10 = 0.02;

double rateTTK = 5;

double rateTTX = 18;
double rateTMTX = 18;

double prbtVPN = 0.8;
double prbtBE = 0.2;

double prbtRT[128];
double my_prbtRT[128];

char nameFP[128];
double FPprecision = 0.000001;

int TIpolicy = 1;
int TIp3_th = 1;

char linktype[128];

int ntoken = 0;

 /* prototype guard functions */
 int guard_tRT () ;
 int guard_tBP () ;
 int guard_tMBP () ;
```

```
  int guard_tBE () ;
  int guard_tVPN () ;
  int guard_tMWT () ;

  /* prototype rate functions */
  double rate_TARV () ;

  /* prototype cardinality functions */
  int cardinality_MWT () ;

void LoadConfig()
{
  FILE *fp;
  char sstr[100], *ptr;
  int i, nodata = 0;

  sprintf(sstr, "%s.cfg", modelname);
  if ((fp = fopen(sstr, "r")) == NULL)
    {
      perror("Can not Open the Config File");
      exit(1);
    }

  while (fgets(sstr, 100, fp)) {
    if (*sstr == '#' || *sstr == '\n')
      continue;

    if (!strncmp(sstr, "Size of Buffer:", 15))
      {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%d", &NBUF);
    continue;
      }

    if (!strncmp(sstr, "Size of Leaky-Bucket Token Pool:", 32))
      {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%d", &NTK);
    continue;
      }

    if (!strncmp(sstr, "Round Robin Weight:", 18))
      {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%d", &NRR);
    continue;
      }

    if (!strncmp(sstr, "Maximum Waiting Time:", 21))
      {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%d", &NMRR);
    continue;
      }

    if (!strncmp(sstr, "Rate of MMPP 0 to 1:", 20))
      {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%lf", &rateTMMPP01);
    continue;
      }

    if (!strncmp(sstr, "Rate of MMPP 1 to 0:", 20))
```

```
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &rateTMMPP10);
continue;
  }

if (!strncmp(sstr, "Leaky-Bucket Token Rate:", 24))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &rateTTK);
continue;
  }

if (!strncmp(sstr, "High Packet Arrival Rate:", 24))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &harvrate);
continue;
  }

if (!strncmp(sstr, "Low Packet Arrival Rate:", 23))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &larvrate);
continue;
  }

if (!strncmp(sstr, "Packet Leaving Rate:", 20))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &rateTTX);
rateTMTX = rateTTX;
continue;
  }

if (!strncmp(sstr, "Percent of High Priority Packet:", 32))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &prbtVPN);
prbtVPN / 100;
continue;
  }

if (!strncmp(sstr, "Percent of Low Priority Packet:", 31))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%lf", &prbtBE);
prbtBE / 100;
continue;
  }

if (!strncmp(sstr, "Erlang Stage:", 13))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%d", &NER);
continue;
  }

if (!strncmp(sstr, "Traffic Interaction Policy:", 27))
  {
ptr = strchr(sstr, ':');
sscanf(ptr + 1, "%d", &TIpolicy);
continue;
```

```
    }

  if (!strncmp(sstr, "Policy III Threshold:", 20))
    {
  ptr = strchr(sstr, ':');
  sscanf(ptr + 1, "%d", &TIp3_th);
  continue;
    }

  if (!strncmp(sstr, "Fix Point Iteration File:", 25))
    {
  ptr = strchr(sstr, ':');
  sscanf(ptr + 1, "%s", nameFP);
  continue;
    }

  if (!strncmp(sstr, "Fix Point Precision:", 20))
    {
  ptr = strchr(sstr, ':');
  sscanf(ptr + 1, "%lf", &FPprecision);
  continue;
    }

  if (!strncmp(sstr, "Link Type:", 10))
    {
  ptr = strchr(sstr, ':');
  sscanf(ptr + 1, "%s", linktype);
  continue;
    }
}

fclose(fp);
rateTTK *= NER;

nodata = 0;
sprintf(sstr, "%s.itp", modelname);
if ((fp = fopen(sstr, "r")) ≠ NULL)
  {
    for (i = 0; i ≤ NRR; i++)
  {
    if (!fgets(sstr, 100, fp))
      {
        nodata = 1;
        break;
      }
    if (*sstr == '#' || *sstr == '\n')
      {
        printf("Error in Iteration Date\n");
        exit(1);
      }
    sscanf(sstr, "%lf", my_prbtRT + i);
  }
    fclose(fp);
  }
else
  nodata = 1;
if (nodata)
  for (i = 0; i ≤ NRR; i++)
    my_prbtRT[i] = 1 / ((double)NRR + 1.0);

nodata = 0;
if ((fp = fopen(nameFP, "r")) ≠ NULL)
```

```
    {
     for (i = 0; i ≤ NMRR; i++)
    {
     if (!fgets(sstr, 100, fp))
       {
        nodata = 1;
        break;
       }
     if (*sstr == '#' || *sstr == '\n')
       {
        printf("Error in Iteration Date\n");
        exit(1);
       }
     sscanf(sstr, "%lf", prbtRT + i);
    }
     fclose(fp);
    }
  else
    nodata = 1;
  if (nodata)
    for (i = 0; i ≤ NMRR; i++)
      prbtRT[i] = 1 / ((double)NMRR + 1.0);


}

void options() {
  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_YES) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;
  /* NUMERICAL SOLUTION chosen */
  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_GASEI) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,20000) ;
  fopt(FOP_PRECISION,0.000001) ;
  fopt(FOP_ABS_RET_M0,0.000000) ;
  iopt(IOP_CUMULATIVE,VAL_YES) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;

  LoadConfig();

}

/* GUARD Functions */
int guard_rRT ()
{
```

```
 if (mark("pHQ") && mark("pTK"))
   return(1);
 else
   return(0);
}
int guard_tBP ()
{
 if (mark("pHQ") && mark("pTK"))
   return(0);
 else
   return(1);
}
int guard_tMBP ()
{
 if (mark("pHQ") && mark("pTK"))
   return(1);
 else
   return(0);
}
int guard_tBE_p12 ()
{
 return(1);
}
int guard_tBE_p3 ()
{
 if (mark("pTK") − mark("pHQ") ≥ TIp3_th)
   return(0);
 else
   return(1);
}
int guard_tVPN ()
{
 if (mark("pHQ") < NBUF)
   return(1);
 else
   return(0);
}
int guard_tMWT ()
{
 if (!mark("pMTX") && mark("pMWT"))
   return(1);
 else
   return(0);
}


/* RATE Functions */
double rate_TARV ()
{
 if (mark("pMMPP"))
   return(harvrate);
 else
   return(larvrate);
}


/* CARDINALITY Functions */
int cardinality_MWT ()
{
 return(mark("pMWT"));
}
```

```
void net() {
  int i;
  char rt[128];

  /* MMPP */
  place("pMMPP");

  rateval("TMMPP10",rateTMMPP10);
  rateval("TMMPP01",rateTMMPP01);

  iarc("TMMPP10","pMMPP");
  oarc("TMMPP01","pMMPP");
  harc("TMMPP01","pMMPP");

  /* Tagged / Aggregated link */
  place("pARV");
  place("pHQ");
  place("pTK");
  place("pER");

  ratefun("TARV",rate_TARV);
  rateval("TTK",rateTTK);
  rateval("TTX",rateTTX);
  imm("tVPN");
  guard("tVPN",guard_tVPN);
  priority("tVPN",20);
  probval("tVPN",prbtVPN);
  imm("tBE");
  if (TIpolicy < 3)
    guard("tBE",guard_tBE_p12);
  else
    guard("tBE",guard_tBE_p3);
  priority("tBE",20);
  probval("tBE",prbtBE);
  imm("tER");
  priority("tER",20);
  probval("tER",1.);

  oarc("TARV","pARV");
  iarc("tBE","pARV");
  iarc("tVPN","pARV");
  oarc("tVPN","pHQ");
  oarc("TTK","pER");
  miarc("tER","pER",NER);
  oarc("tER","pTK");
  mharc("TTK","pTK",NTK);
  iarc("TTX","pHQ");
  iarc("TTX","pTK");

  /* Round Robin */
  place("pRR");
  place("pRC");
  init("pRC",NRR);
  place("pMTX");
  place("pMWT");

  rateval("TMTX",rateTMTX);

  imm("tBP");
  guard("tBP",guard_tBP);
  priority("tBP",20);
  probval("tBP",1.);
  imm("tMWT");
  guard("tMWT",guard_tMWT);
  priority("tMWT",20);
```

```
  probval("tMWT",1.) ;
  imm("tMBP") ;
  guard("tMBP",guard_tMBP) ;
  priority("tMBP",20) ;
  probval("tMBP",prbtRT[0]) ;

  oarc("TTX","pRC") ;
  iarc("TTX","pRR") ;
  iarc("tBP","pRR") ;
  oarc("tBP","pRC") ;
  miarc("tMBP","pRC",NRR) ;
  moarc("tMBP","pRR",NRR) ;
  iarc("TMTX","pMTX") ;
  oarc("TMTX","pMWT") ;
  viarc("tMWT","pMWT",cardinality_MWT) ;
  moarc("tMWT","pRR",NRR) ;

  for (i = 1; i ≤ NMRR; i++)
    {
      sprintf(rt, "tRT_%d", i);
      imm(rt) ;
      guard(rt,guard_tRT) ;
      priority(rt,20) ;
      probval(rt,prbtRT[i]) ;
      miarc(rt,"pRC",NRR) ;
      moarc(rt,"pMTX",i) ;
    }
}

int assert()
{
  return RES_NOERR;
}

void ac_init()
{
}

void ac_reach()
{
}

double get_prbtRT()
{
  if (((mark("pTK") == ntoken) && (mark("pHQ") ≥ ntoken))
      || ((mark("pTK") ≥ ntoken) && (mark("pHQ") == ntoken)))
    return(1.0);
  else
    return(0.0);
}

double get_though()
{
  if (mark("pHQ") < NBUF)
    {
      if (mark("pMMPP"))
    return(prbtVPN * harvrate);
      else
    return(prbtVPN * larvrate);
    }
  else
    return(0.0);
}
```

```
double get_arrival()
{
 if (mark("pMMPP"))
   return(harvrate);
 else
   return(larvrate);
}

double get_util()
{
 if (mark("pTK") < NTK)
   return(1.0);
 else
   return(0.0);
}

double get_VPN2BE()
{
 if (mark("pHQ") == NBUF)
   {
    if (mark("pMMPP"))
   return(prbtVPN * harvrate);
     else
   return(prbtVPN * larvrate);
   }
 else
   return(0.0);
}

double get_prbVPN2BE()
{
 if (mark("pHQ") == NBUF)
   return(1.0);
 else
   return(0.0);
}

double get_prbBVPN2BE()
{
 if ((mark("pHQ") == NBUF) && mark("pMMPP"))
   return(1.0);
 else
   return(0.0);
}

double get_prbNBVPN2BE()
{
 if ((mark("pHQ") == NBUF) && !mark("pMMPP"))
   return(1.0);
 else
   return(0.0);
}

double get_BE2VPN()
{
 if (mark("pTK") − mark("pHQ") ≥ TIp3_th)
   {
    if (mark("pMMPP"))
   return(prbtBE * harvrate);
     else
   return(prbtBE * larvrate);
   }
```

```
  else
    return(0.0);
}

double get_prbBE2VPN()
{
  if (mark("pTK") − mark("pHQ") ≥ TIp3_th)
    return(1.0);
  else
    return(0.0);
}

double get_prbBBE2VPN()
{
  if ((mark("pTK") − mark("pHQ") ≥ TIp3_th) && mark("pMMPP"))
    return(1.0);
  else
    return(0.0);
}

double get_prbNBBE2VPN()
{
  if ((mark("pTK") − mark("pHQ") ≥ TIp3_th) && !mark("pMMPP"))
    return(1.0);
  else
    return(0.0);
}


double get_prbBEMPTY()
{
  if ((!mark("pTK") || !mark("pHQ") || !mark("pRR")) && mark("pMMPP"))
    return(1.0);
  else
    return(0.0);
}
double get_prbNBEMPTY()
{
  if ((!mark("pTK") || !mark("pHQ") || !mark("pRR")) && !mark("pMMPP"))
    return(1.0);
  else
    return(0.0);
}

double get_prbBust()
{
  if (mark("pMMPP"))
    return(1.0);
  else
    return(0.0);
}

void ac_final()
{
  double delta = 0.0, tmp_prbtRT, maxvalue, total_prbtRT = 0.0;
  double prbVPN2BE_h, prbVPN2BE_l, prbBE2VPN_h = 0, prbBE2VPN_l = 0;
  double prbBust;
  FILE *fp;
  char sstr[100];
  int i;

  solve(INFINITY);
```

```
sprintf(sstr, "%s.itp", modelname);
if ((fp = fopen(sstr, "w")) == NULL)
  {
   perror("Can not Open the Fix Point Data File");
   exit(1);
  }

for (ntoken = 0; ntoken < NRR; ntoken++)
  {
   tmp_prbtRT = expected(get_prbtRT);
   maxvalue = max(fabs(my_prbtRT[ntoken]), fabs(tmp_prbtRT));
   my_prbtRT[ntoken] -= tmp_prbtRT;
   maxvalue = fabs(my_prbtRT[ntoken]) / maxvalue;
   if (maxvalue > delta)
  delta = maxvalue;
   total_prbtRT += tmp_prbtRT;
   fprintf(fp, "%.12e\n", tmp_prbtRT);
  }
my_prbtRT[NRR] -= 1.0 - total_prbtRT;
fprintf(fp, "%.12e\n", 1.0 - total_prbtRT);

fclose(fp);

pr_std_average();

if (delta < FPprecision)
  {
   printf("Fix Point Precision Satisfied\n");
   if (!strncmp(linktype, "Tagged", 6))
  {
   prbBust = expected(get_prbBust);
   fprintf(Outfile,"\nProb. of VPN getting through = %.12e\n",
      expected(get_though)/ (prbtVPN * expected(get_arrival)));
   fprintf(Outfile,"\nUltilization of VPN = %.12e\n", expected(get_util));
   prbVPN2BE_h = expected(get_prbBVPN2BE) / prbBust;
   prbVPN2BE_l = expected(get_prbNBVPN2BE) / (1 - prbBust);
   fprintf(Outfile,"\nTraffic from VPN to Best-Effort = %.12e\n",
      expected(get_VPN2BE));
   fprintf(Outfile,"\nProb. of Traffic from VPN to Best-Effort = %.12e\n",
      expected(get_prbVPN2BE));
   fprintf(Outfile,"\nProb. of Traffic from VPN to Best-Effort at Busty = %.12e\n",
      prbVPN2BE_h);
   fprintf(Outfile,"\nProb. of Traffic from VPN to Best-Effort at Non-Busty = %.12e\n"
      , prbVPN2BE_l);
   if (TIpolicy == 3)
     {
      fprintf(Outfile,"\nTraffic from Best-Effort to VPN = %.12e\n",
         expected(get_BE2VPN));
      prbBE2VPN_h = expected(get_prbBBE2VPN) / prbBust;
      prbBE2VPN_l = expected(get_prbNBBE2VPN) / (1 - prbBust);
      fprintf(Outfile,"\nProb. of Traffic from Best-Effort to VPN = %.12e\n",
         expected(get_prbBE2VPN));
      fprintf(Outfile,"\nProb. of Traffic from Best-Effort to VPN at Busty = %.12e\n"
         , prbBE2VPN_h);
      fprintf(Outfile,"\nProb. of Traffic from Best-Effort to VPN at Non-Busty = %.12e\n",
         prbBE2VPN_l);
     }

   sprintf(sstr, "%s.pda", modelname);
   if ((fp = fopen(sstr, "w")) == NULL)
     {
      perror("Can not Open Data File");
      fclose(Outfile);
      exit(1);
```

```
    }
  for (i = 1; i ≤ 3; i++)
    {
     switch(i)
    {
    case 1:
     fprintf(fp, "%.12e\t%.12e\n", expected(get_prbNBEMPTY) /
        (1 − prbBust), expected(get_prbBEMPTY) / prbBust);
     break;
    case 2:
     fprintf(fp, "%.12e\t%.12e\n", prbVPN2BE_l, prbVPN2BE_h);
     break;
    case 3:
     fprintf(fp, "%.12e\t%.12e\n", prbBE2VPN_l, prbBE2VPN_h);
     break;
    }
    }
  fclose(fp);
  fclose(Outfile);
  exit(2);
 }
 }
}
```

3. Config file *smod3ta.cfg*

```
# link type
Link Type: Aggregated

# MMPP
Rate of MMPP 0 to 1: 1
Rate of MMPP 1 to 0: 1

# tagged / aggregated links
Size of Buffer: 224
Size of Leaky−Bucket Token Pool: 112
Erlang Stage: 1
Leaky−Bucket Token Rate: 63
High Packet Arrival Rate: 107.3431
Low Packet Arrival Rate: 67.6569
Percent of High Priority Packet: 90
Percent of Low Priority Packet: 10
Packet Leaving Rate: 100

# round robin
Round Robin Weight: 7
Maximum Waiting Time: 1

# traffic interation policy
Traffic Interaction Policy: 3
Policy III Threshold: 1

# fix point iteration
Fix Point Iteration File: smod3ta.itp
Fix Point Precision: 0.0001
```

10.13 Simulation example: reader and writer sharing buffer

10.13.1 Source

Computer Science Department. College of Willian and Mary. *On the Simulation of Stochastic Petri Nets*.

The parameters have been changed to have rare events.

10.13.2 Description

The net shown in Figure 10.18 represents a set of N operating processes in an operating system sharing a buffer in sharing or writing mode. Up to $K \leq N$ processes are allowed to read the buffer. Once a process is in the ready-to-write state, the operating system informs the other processes that are ready to read that they should wait because the buffer will be modified. In the same way, all the processes that already are in the reading state should exit at once because the data is out of date.

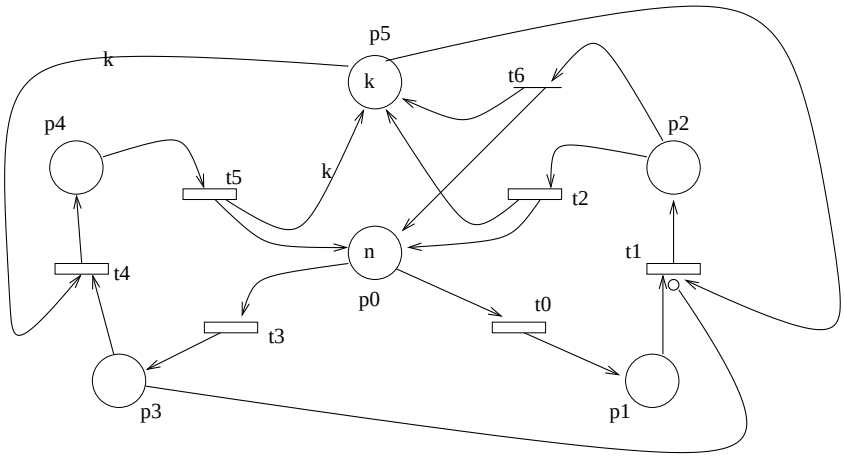


Figure 10.18: Reader and Writer sharing buffer

10.13.3 Features

- Involves non-exponential distributions.
- Use of discrete-event simulation.

- Use of resampling policies.

10.13.4 SPNP File — *readwrite.c*

```
/*******************************************************/
/*Title: A example of reader and writer sharing buffer */
/*File : readerwriterbuffer.c                          */
/*Time : Jul-04-97                                     */
/*******************************************************/
#include "user.h"
int n,k;

void options() {
      iopt(IOP_SIMULATION,VAL_YES);
   iopt(IOP_SIM_CUMULATIVE,VAL_YES);
   iopt(IOP_SIM_STD_REPORT,VAL_YES);
      //iopt(IOP_SIM_SEED,345983453);
      iopt(IOP_SIM_RUNS,100000);
      fopt(FOP_SIM_LENGTH,20.);
      fopt(FOP_SIM_CONFIDENCE,.95);

   n=input("Total Processes");
   k=input("Max Reading Processes");
}

int fun() { return ( mark("p2") );}
int gun() { return ( (mark("p3")>0 && mark("p2")>0) ? 1: 0) ;}

void net(){
 place("p0"); place("p1"); place("p2"); place("p3"); place("p4"); place("p5");
 init("p0",n); init("p5",k);

 rateval("t0",4.0);  unifval("t1",1.0,2.0);  normval("t2",2.0,0.5);
 rateval("t3",1.0);  unifval("t4",1.0,2.0);  normval("t5",3.0,1.0);
 imm("t6");  guard("t6",gun);

 policy("t1",PRS);policy("t2",PRD);
 iarc("t0","p0");  oarc("t0","p1");
 iarc("t1","p1");  iarc("t1","p5"); oarc("t1","p2"); harc("t1","p3");
 iarc("t2","p2");  oarc("t2","p0");  oarc("t2","p5");
 iarc("t3","p0");  oarc("t3","p3");
 iarc("t4","p3");  miarc("t4","p5",k);  oarc("t4","p4");
 iarc("t5","p4");  oarc("t5","p0");   moarc("t5","p5",k);
 viarc("t6","p2",fun); voarc("t6","p5",fun);  voarc("t6","p0",fun);
}

void ac_init() { pr_net_info();}

int assert() { return (1); }

void ac_reach() {}

double eff() { return((mark("p2")≥6)?1:0); }

void ac_final() {
 pr_cum_expected("More than in p2.",eff);
}
```

10.14 Hybrid System: reactor temperature control system

10.14.1 Source

B. Tuffin, D.S. Chen and K.S. Trivedi. “Comparison of Hybrid Systems and Fluid Stochastic Petri Nets”, Technical Report. *Center for Advanced Computing and Communication, Duke University*, 1999.

10.14.2 Description

The net is shown in Figure 10.19. It represents a reactor temperature control system. The reactor core temperature rises at a linear rate. To control the temperature, a rod, chosen randomly between two, is put into the reactor core when the temperature reaches 550 degrees. The rod is then removed when the temperature falls back to 510 degrees

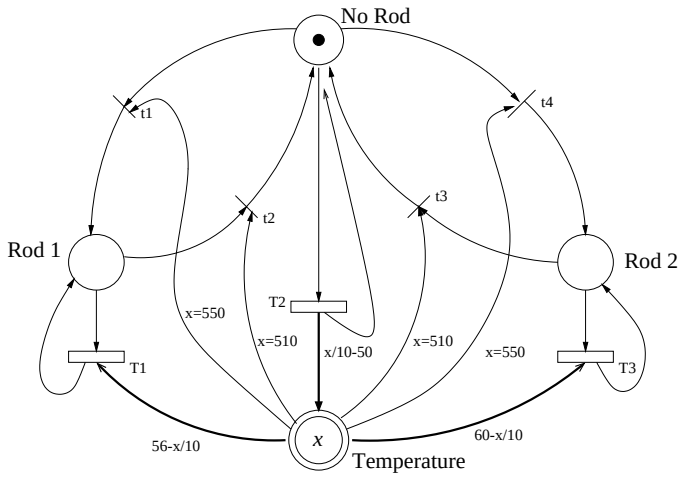


Figure 10.19: FSPN model of reactor temperature control example

10.14.3 Features

- This is an FSPN.
- The flows are marking dependent and linear.

10.14.4 SPNP File — *reactor.c*

```
/* This example is from "Comparison of FSPNs and HS                    */
/*  –by Bruno Tuffin el.                                               */
/***************************************************************************/

#include  "user.h"

int c3,c4;

void options() {
      iopt(IOP_SIMULATION,VAL_YES);
   iopt(IOP_SIM_CUMULATIVE,VAL_YES);
   iopt(IOP_SIM_STD_REPORT,VAL_YES);
      iopt(IOP_SIM_SEED,345983453);
      iopt(IOP_SIM_RUNS,10000);
      fopt(FOP_SIM_LENGTH,1000.);
      fopt(FOP_SIM_CONFIDENCE,.95);
      fopt(FOP_FLUID_EPSILON,0.00000001);
}

double f() {return(0.1);}
double fp() {return(−0.1);}
double g1() {return(−50.0);}
double g2() {return(56.0);}
double g3() {return(60.0);}

int g_decrease() {c3=fcondition("Temperature",F_GE,550.0); return(c3);}
int g_increase() {c4=fcondition("Temperature",F_LQ,510.0); return(c4);}

void net() {
 place("No_rod"); place("Rod_1"); place("Rod_2");init("No_rod",1);
 fplace("Temperature"); finit("Temperature",512.0);
 fbound("Temperature",550.0);
 fbreak("Temperature",510.0);
 inf("T2"); iarc("T2","No_rod"); oarc("T2","No_rod"); floarc("T2","Temperature",f,g1);
 inf("T1"); iarc("T1","Rod_1"); oarc("T1","Rod_1"); fliarc("T1","Temperature",fp,g2);
 inf("T3"); iarc("T3","Rod_2"); oarc("T3","Rod_2"); fliarc("T3","Temperature",fp,g3);
 imm("t1");      probval("t1",1.0);  guard("t1",g_decrease);
 imm("t2");     probval("t2",1.0);  guard("t2",g_increase);
 imm("t4");      probval("t4",1.0);  guard("t4",g_decrease);
 imm("t3");     probval("t3",1.0);  guard("t3",g_increase);
 iarc("t1","No_rod");oarc("t1","Rod_1");
 iarc("t2","Rod_1");oarc("t2","No_rod");
 iarc("t3","Rod_2");oarc("t3","No_rod");
 iarc("t4","No_rod");oarc("t4","Rod_2");

}

void ac_init() { pr_net_info(); }

int assert() { return (1); }

void ac_reach() {}

void ac_final() {
}
```

10.15 Dual tank example

10.15.1 Source

G. Ciardo, D.M. Nicol, and K.S. Trivedi. Discrete-Event Simulation of Fluid Stochastic Petri-Nets. *IEEE Transactions on Software Engineering*, 25(2):207–217, 1999.

The parameters have been changed to have rare events.

10.15.2 Description

The system described in Figure 10.20 contains two tanks "One" and "Two" (of respective maximal capacity b_1 and b_2). A fluid flows with rate r_{in} from an external source into tank "One", which sends the liquid to a processing station with rate $r_{out} > r_{in}$. The processing station is subject to breakdowns and repairs (exponentially distributed with respective rates λ and μ). During a breakdown, the station is not fuelled and the liquid from the external source is immediatley redirected to the additional tank "Two". The external source is shut down only when tank "Two" is full. When the processing station is repaired, the external flow is immediately switched to tank "One" which resume his work. In addition, the liquid in tank "Two" is pumped into tank "One" with rate r_{21}. If tank One is full, the flow from tank "two" to tank "One" is slowed.

10.15.3 Features

- This is an FSPN.
- The flows are marking dependent.
- Computation of the probability to have a full tank "Two".

10.15.4 SPNP File — *splitting.c*

```
/* This example is adapted from "Discrete-event simulation of fluid stochastic */
/*Petre nets"      –by Gianfranco Ciardo   el.                                  */
```

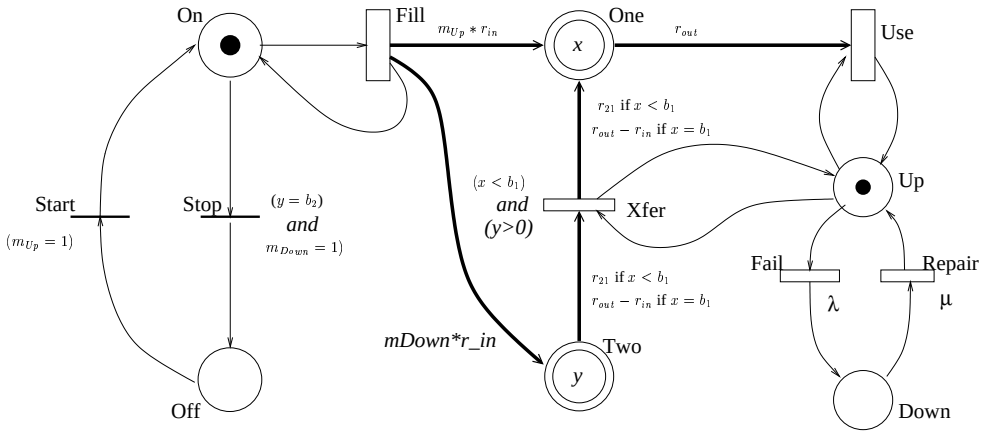


Figure 10.20: FSPN model of a dual tank

```
/*************************************************************************/

#include  "user.h"

#define bound1 1.0
#define bound2 2.0
#define rin    0.08
#define r21    0.97
#define rout   1

int c1,c2,c3,c4;

void options() {
      iopt(IOP_SIMULATION,VAL_YES);
      iopt(IOP_SIM_SPLIT,VAL_YES);
      iopt(IOP_SIM_SPLIT_RESTART,VAL_YES);
      iopt(IOP_SPLIT_PRESIM,VAL_YES);
      iopt(IOP_SPLIT_PRESIM_RUNS,1000);
      iopt(IOP_SPLIT_RESTART_FINISH,VAL_NO);
   iopt(IOP_SIM_CUMULATIVE,VAL_NO);
   iopt(IOP_SIM_STD_REPORT,VAL_YES);
      iopt(IOP_SIM_SEED,345983453);
      iopt(IOP_SIM_RUNS,0);
      fopt(FOP_SIM_ERROR,0.1);
      fopt(FOP_SIM_LENGTH,100.);
      fopt(FOP_SIM_CONFIDENCE,.95);
      fopt(FOP_FLUID_EPSILON,0.00000001);
      fopt(FOP_TIME_EPSILON,0.00000001);

      /* the following option stands if iopt(IOP_SIM_SPLIT_RESTART,VAL_YES);*/
      iopt(IOP_SPLIT_LEVEL_DOWN,4);

      /* the following options stand if iopt(IOP_SPLIT_PRESIM,VAL_NO); */
      iopt(IOP_SPLIT_NUMBER,6);
      FOP_SPLIT_THRESHOLDS[1]=0.429032;
      FOP_SPLIT_THRESHOLDS[2]=0.666285;
      FOP_SPLIT_THRESHOLDS[3]=0.095069;
      FOP_SPLIT_THRESHOLDS[4]=1.161825;
      FOP_SPLIT_THRESHOLDS[5]=1.376134;
```

```
      FOP_SPLIT_THRESHOLDS[6]=1.595820;
}

double fuprin() {return(rin*mark("Up"));}
double fdnrin() {return(rin*mark("Down"));}
double f21()
 { double val;
   c4=fcondition("Two",F_GT,0);
   c1=fcondition("One",F_LT,bound1);
   val=c4 ? (c1 ? r21 :(rout-rin)) : 0;
   return (val);
 }
double fout()
 { c4=fcondition("Two",F_GT,0);
   c2=fcondition("One",F_GT,0);
   return (c4 ? rout: (c2? rout: rin));
 }


int gstop() {c3=fcondition("Two",F_EQ,bound2); return(c3*mark("Down"));}
int gstart(){return(mark("Up"));}

void net() {
  place("On"); place("Off"); place("Up"); place("Down");init("On",1); init("Up",1);
  fplace("One"); finit("One",0.0); fbound("One",bound1);
  fplace("Two"); fbound("Two",bound2);
  inf("Fill");
  inf("Use");
  inf("Xfer");
  imm("Stop");       probval("Stop",1.0);   guard("Stop",gstop);
  imm("Start");      probval("Start",1.0);  guard("Start",gstart);
  rateval("Fail",0.1);
  rateval("Repair",1);

  iarc("Fill","On");oarc("Fill","On");
     fvoarc("Fill","One",fuprin); fvoarc("Fill","Two",fdnrin);

  iarc("Stop","On");
     oarc("Stop","Off");

  iarc("Start","Off"); oarc("Start","On");

  iarc("Xfer","Up"); oarc("Xfer","Up");
    fviarc("Xfer","Two",f21);fvoarc("Xfer","One",f21);

  iarc("Use","Up"); oarc("Use","Up"); fviarc("Use","One",fout);

  iarc("Fail","Up");oarc("Fail","Down");iarc("Repair","Down");oarc("Repair","Up");
}

void ac_init() { pr_net_info(); }

int assert() { return (1); }

void ac_reach() {}

double eff() { return((fmark("Two")>2.0) ? 1:0); }

void ac_final() {
  splitting("Two", 2.0);
}
```

10.16 Equivalent failure rate and repair rate computation in hierarchical model

10.16.1 Source

M. Lanus, L. Yin, "Bedrock Availability Analysis", In *Proc. Motorola SES99, Software Engineering Symposium*, June 1999. (This source might not be available to outside Motorola. The system configuration, data and code shown in the example have been modified due to confidential reason.)

M. Lanus, L. Yin, K. S. Trivedi, "Hierarchical Decomposition and Aggregation of State-based Availability and Performability Models for Telecommunications Systems", submitted to *IEEE Transactions on Reliability*.

10.16.2 Description

The Markov chain shown in Figure 10.21 models a sub-system module which has 1+1 hot standby redundancy. It models the details of hardware/software failure of active/standby component, switch over, coverage, reboot, and repair. Due to the complexity, it is easier to use Markov chain rather than SPN. However, any Markov chain can be implemented in SPN by simply converting states into places, transitions into SPN transitions and arcs.

The system level SPN model in Figure 10.22 has a 2-state Markov chain (2-place 2-transition SPN) for each sub-system module. The equivalent failure rate and repair rate (MTTF and MTTR) are computed at lower level model and fed into the system level model. Compared to the brutal force approach, this hierarchical approach largely reduces the number of states at the system level. This approach is an exact technique (not an approximation) for steady state measurement like Downtime Performance Measurement (DPM) as defined in Bellcore RQSM.

The interaction of system level model and lower level model is done by system calls

and parsing the output file of lower level model. The computation of equivalent rates is done by partitioning the states into Up states and Down states. The key functions are `hold_cond()` and `pr_value()`.

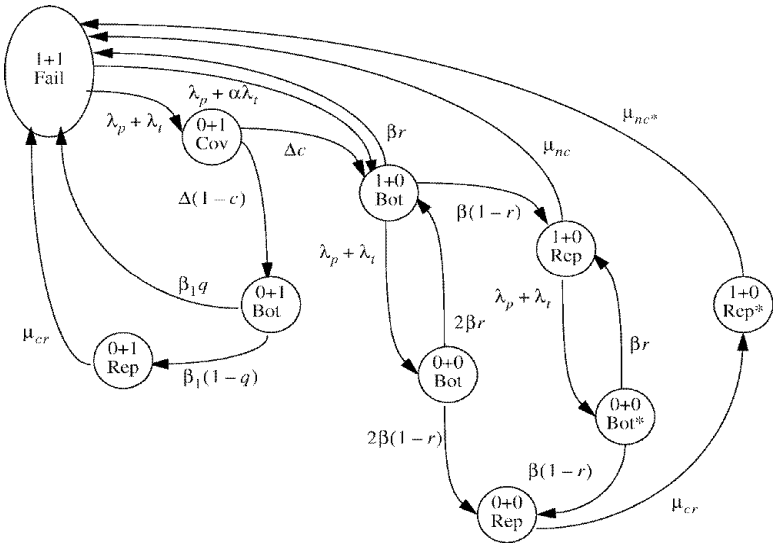


Figure 10.21: Markov Chain model for 1+1 (hot standby) redundancy modules

10.16.3 Features

- Computation of equivalent failure rate and repair rate
- Hierarchical Model
- Markov chain modeling details of 1+1 (hot standby) redundancy
- Availability measurement – Downtime Performance Measurement (DPM)

10.16.4 SPNP Files for Module level Markov chain model

HSP module — *gsbhsp.c*

```
#include <stdio.h>
#include "user.h"

/* global variables */

/* all the rates are in the format 1.0/MTTF or 1.0/MTTR (unit: 1/Hour) */

double lambdaP=1.0/180000.0;   /* (hardware) failure rate for HSP */
double lambdaT=1.0/4000.0;   /* transient (/sw) failure rate for HSP */
```

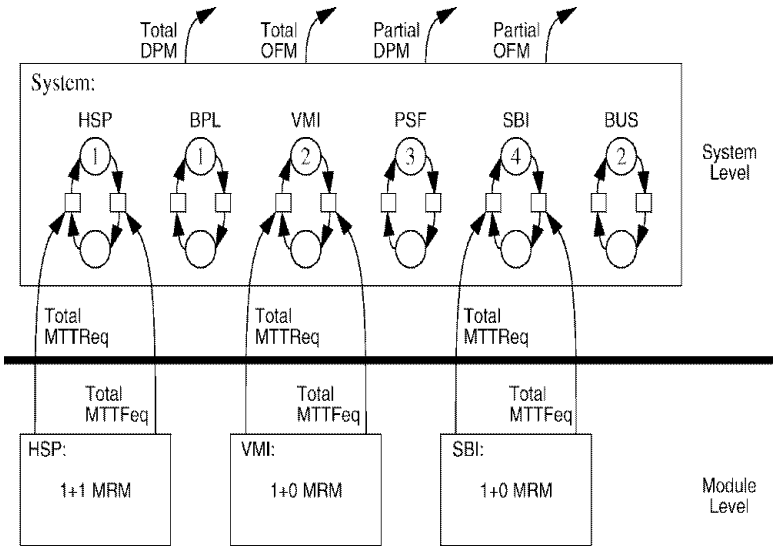


Figure 10.22: SPNP model for the system

```
double muCr=1.0/4.0;       /* repair rate for Critical HSP fault */
double muNc=1.0/12.0;      /* repair rate for Non-Critical HSP fault */

/* repair rate for non-critical HSP fault (after having repaired critical */
/*       HSP fault. */
double muNcS=1.0;

/* reboot time for HSP fault (active HSP failed, unsuccessful detection, */
/*       standby HSP working),  1 min */
double beta1=1.0/(1.0/60.0);

/* reboot time for HSP fault (???) */
double beta=1.0/(1.0/60.0);

/* Active HSP fault detection/switching to standby HSP delay time,  5 sec */
double delta=1.0/(4.0/3600.0);

/* coverage for the above detection/switching */
double c=0.98;

/* coverage for reboot (beta1) */
double q=0.80;

/* coverage for reboot (beta) */
double r;

/* Fraction of transient/sw faults that are manifest in Standby HSP */
double alpha=0.2;

/* threshold for counting the unavailability of the HSPs */
double tR=10.0/3600.0;

/* prototype reward functions */
double dtime();
double dtimeFC();
double dtimeF();

/* prototype guard functions */
```

```
/* prototype rate functions */
double rHsp2_D();
double rHsp2_hsp1();
double rD_hsp1();
double rD_Ds();
double rDs_hsp2();
double rDs_FC();
double rHsp1_hsp1s();
double rHsp1_hsp2();
double rHsp1_hsp0();
double rHsp0_F();
double rHsp0_hsp1();
double rHsp1s_hsp0s();
double rHsp0s_hsp1s();
double rHsp0s_F();

/* values for global variables */

/* prototype cardinality functions */



void options() {
  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;
  /* NUMERICAL SOLUTION chosen */
  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,2000) ;
  fopt(FOP_PRECISION,0.000001) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_NO) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;

  /* some computation */
  r = lambdaT/(lambdaP+lambdaT);

}


/* REWARD Functions */
double dtime() {
  double dtime;
  dtime = ( (double)mark("Ds")*exp(-beta1*tR)
```

```
        + (double)mark("D")*exp(-delta*tR)
        + (double)mark("hsp0s")*exp(-beta*tR)
        + (double)mark("hsp0")*exp(-beta*tR)
        + (double)mark("FC")*1.0
        + (double)mark("F")*1.0 );
 dtime *= 8766*60;
 return dtime;
}

double dtime60vs10() {
 double dtime;
 dtime = ( (double)mark("Ds")
        + (double)mark("D")*exp(-delta*tR)
        + (double)mark("hsp0s")
        + (double)mark("hsp0")
        + (double)mark("FC")*1.0
        + (double)mark("F")*1.0 );
 dtime *= 8766*60;
 return dtime;
}

double dtime5vs10() {
 double dtime;
 dtime = ( (double)mark("Ds")
        + (double)mark("hsp0s")
        + (double)mark("hsp0")
        + (double)mark("FC")*1.0
        + (double)mark("F")*1.0 );
 dtime *= 8766*60;
 return dtime;
}

int cond() {
 if ( mark("hsp2")==0 && mark("hsp1")==0 && mark("hsp1s")==0
     && mark("hsp1ss")==0 )  { return 1; }
 else { return 0; }
}

double times[2];

int condFC() {
 if ( mark("FC")==1)  { return 1; }
 else { return 0; }
}

double timesFC[2];

int condF() {
 if ( mark("F")==1)  { return 1; }
 else { return 0; }
}

double timesF[2];

int cond5vs10() {
 if ( mark("hsp2")==0 && mark("hsp1")==0 && mark("hsp1s")==0
     && mark("hsp1ss")==0 && mark("D")==0 )  { return 1; }
 else { return 0; }
}

double times5vs10[2];
```

```
double tsodpm() {
  double dtime;
  dtime = ( (double)mark("Ds")
        + (double)mark("D")
        + (double)mark("hsp0s")
        + (double)mark("hsp0")
        + (double)mark("FC")
        + (double)mark("F"));
  dtime *= 8766*60;
  return dtime;
}

double dtimeFC() {
  double dtimeFC;
  dtimeFC = (double)mark("FC")*1.0;
  dtimeFC *= 8766*60;
  return dtimeFC;
}

double dtimeF() {
  double dtimeF;
  dtimeF = (double)mark("F")*1.0;
  dtimeF *= 8766*60;
  return dtimeF;
}

/* GUARD Functions */

/* RATE Functions */
double rHsp2_D() {
  return(lambdaP+lambdaT);
}
double rHsp2_hsp1() {
  return(lambdaP+alpha*lambdaT);
}
double rD_hsp1() {
  return(delta*c);
}
double rD_Ds() {
  return(delta*(1-c));
}
double rDs_hsp2() {
  return(beta1*q);
}
double rDs_FC () {
  return(beta1*(1-q));
}
double rHsp1_hsp1s() {
  return(beta*(1-r));
}
double rHsp1_hsp2() {
  return(beta*r);
}
double rHsp1_hsp0() {
  return(lambdaP+lambdaT);
}
double rHsp0_F() {
  return(beta*(1-r));
```

```
}
double rHsp0_hsp1() {
  return(beta*r);
}
double rHsp1s_hsp0s() {
  return(lambdaP+lambdaT);
}
double rHsp0s_hsp1s() {
  return (beta*r);
}
double rHsp0s_F() {
  return (beta*(1-r));
}

/* CARDINALITY Functions */

void net() {

  /* PLACE */       /* Active HSP          Standby HSP */
  place("hsp2");    /* no fault          no fault   */
  init("hsp2",1);
  place("D");       /* fault, try to detect       no faul   */
  place("Ds");      /* fault not covered, reboot   no fault   */
  place("FC");      /* reboot failed, repair   no faul   */
  place("hsp1");    /* no fault          reboot   */
  place("hsp1s");   /* no fault          repair   */
  place("hsp1ss");  /* no faul           repair/reboot */
  place("hsp0");    /* reboot          waiting for reboot */
  place("hsp0s");   /* reboot          waiting for repair */
  place("F");       /* repair          waiting for repair */
  /* TRANSITION */
  ratefun("hsp2_D",rHsp2_D);
  ratefun("hsp2_hsp1",rHsp2_hsp1);
  ratefun("D_hsp1",rD_hsp1);
  ratefun("D_Ds",rD_Ds);
  ratefun("Ds_hsp2",rDs_hsp2);
  ratefun("Ds_FC",rDs_FC);
  rateval("FC_hsp2",muCr);
  ratefun("hsp1_hsp1s",rHsp1_hsp1s);
  ratefun("hsp1_hsp2",rHsp1_hsp2);
  ratefun("hsp1_hsp0",rHsp1_hsp0);
  ratefun("hsp0_F",rHsp0_F);
  ratefun("hsp0_hsp1",rHsp0_hsp1);
  ratefun("hsp1s_hsp0s",rHsp1s_hsp0s);
  rateval("hsp1s_hsp2",muNc);
  ratefun("hsp0s_hsp1s",rHsp0s_hsp1s);
  ratefun("hsp0s_F",rHsp0s_F);
  rateval("F_hsp1ss",muCr);
  rateval("hsp1ss_hsp2",muNcS);
  /* ARC */
  iarc("hsp2_D","hsp2");
  oarc("hsp2_D","D");
  iarc("hsp2_hsp1","hsp2");
  oarc("hsp2_hsp1","hsp1");
  iarc("D_hsp1","D");
  oarc("D_hsp1","hsp1");
  iarc("D_Ds","D");
  oarc("D_Ds","Ds");
  iarc("Ds_hsp2","Ds");
  oarc("Ds_hsp2","hsp2");
  iarc("Ds_FC","Ds");
  oarc("Ds_FC","FC");
```

```
  iarc("FC_hsp2","FC");
  oarc("FC_hsp2","hsp2");
  iarc("hsp1_hsp1s","hsp1");
  oarc("hsp1_hsp1s","hsp1s");
  iarc("hsp1_hsp2","hsp1");
  oarc("hsp1_hsp2","hsp2");
  iarc("hsp1_hsp0","hsp1");
  oarc("hsp1_hsp0","hsp0");
  iarc("hsp0_F","hsp0");
  oarc("hsp0_F","F");
  iarc("hsp0_hsp1","hsp0");
  oarc("hsp0_hsp1","hsp1");
  iarc("hsp1s_hsp0s","hsp1s");
  oarc("hsp1s_hsp0s","hsp0s");
  iarc("hsp1s_hsp2","hsp1s");
  oarc("hsp1s_hsp2","hsp2");
  iarc("hsp0s_hsp1s","hsp0s");
  oarc("hsp0s_hsp1s","hsp1s");
  iarc("hsp0s_F","hsp0s");
  oarc("hsp0s_F","F");
  iarc("F_hsp1ss","F");
  oarc("F_hsp1ss","hsp1ss");
  iarc("hsp1ss_hsp2","hsp1ss");
  oarc("hsp1ss_hsp2","hsp2");
}

void assert() {

}

void ac_init() {

}

void ac_reach() {

}

void ac_final() {
/*
  double t;
  for (t=0; t≤50000; t+=30000)  {
    solve(t);
    pr_expected("dtime",dtime);
    pr_expected("dtimeFC",dtimeFC);
    pr_expected("dtimeF",dtimeF);
  }
*/
  solve(INFINITY);
  pr_expected("dtime",dtime);
  pr_expected("dtimeFC",dtimeFC);
  pr_expected("dtimeF",dtimeF);
  pr_expected("tsodpm",tsodpm);
  pr_hold_cond("test",cond);
  hold_cond(cond,times);
  pr_value("times0",times[0]);
  pr_value("times1",times[1]);
  pr_value("tsodpm computed from times", times[1]/(times[1]+times[0])∗8766∗60);

  pr_message("\nlook at state FC\n");
  hold_cond(condFC,timesFC);
  pr_value("timesFC0",timesFC[0]);
```

```
  pr_value("timesFC1",timesFC[1]);
  pr_value("dtimeFC computed from timesFC", timesFC[1]/(timesFC[1]+timesFC[0])
        *8766*60);

  pr_message("\nlook at state F\n");
  hold_cond(condF,timesF);
  pr_value("timesF0",timesF[0]);
  pr_value("timesF1",timesF[1]);
  pr_value("dtimeF computed from timesF", timesF[1]/(timesF[1]+timesF[0])
        *8766*60);

  pr_message("\nput Ds,0,0s into down state\n");
  pr_expected("dtime60vs10",dtime60vs10);

  pr_message("\n------------------\n");
  pr_message("\n(This is what actually used ");
  pr_message("and fed into upper level.)\n");
  pr_message("\nPut D into up state, Ds,0,0s into down state.\n");
  pr_expected("dtime5vs10",dtime5vs10);
  hold_cond(cond5vs10,times5vs10);
  pr_value("MTTFeq",times5vs10[0]);
  pr_value("MTTReq",times5vs10[1]);
  pr_value("ofm: ", (1-expected(dtime5vs10)/(8766*60)) / times5vs10[0] *8766);

  pr_mc_info();
  pr_std_average();
}
```

VMI and SBI module — *gsbvsc1.c*

```
#include <stdio.h>
#include "user.h"

/* global variables */

/* all the rates are in the format 1.0/MTTF or 1.0/MTTR (unit: 1/Hour) */

double lambdaP=1.0/120000.0;    /* (hardware) failure rate for HSP */
double lambdaT=1.0/4000.0;    /* transient (/sw) failure rate for HSP */

/* reboot time for HSP fault (active HSP failed, unsuccessful detection, */
/*        standby HSP working),   1 min */
double beta1=1.0/(1.0/60.0);

/* coverage for reboot (beta1)  */
/*double q=0.80;*/
double q;

/* coverage for reboot (beta)   */
double r;

/* repair rate of Ps+Fn, Bus, NSP, DSP  */
/*   in lost redudency/partial outage/total outage */
double rrLr=1.0/24.0;
double rrPo=1.0/8.0;
double rrTo=1.0/4.0;


/* prototype reward functions */
```

```
double dtime();
int cond();

/* prototype guard functions */

/* prototype rate functions */
double rHsp2_Ds();
double rDs_hsp2();
double rDs_FC();

/* values for global variables */

/* prototype cardinality functions */



void options() {
  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;
  /* NUMERICAL SOLUTION chosen */
  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,2000) ;
  fopt(FOP_PRECISION,0.000001) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_NO) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;

  /* some computation */
  r = lambdaT/(lambdaP+lambdaT);
  q=r;

}


/* REWARD Functions */

double dtime() {
  double dtime;
  dtime = ( (double)mark("Ds")
        + (double)mark("FC"));
  dtime *= 8766*60;
  return dtime;
}
```

```
int cond() {
  if ( mark("hsp2")==0 )  { return 1; }
  else { return 0; }
}

double times[2];


/* GUARD Functions */


/* RATE Functions */
double rHsp2_Ds() {
  return(lambdaP+lambdaT);
}
double rDs_hsp2() {
  return(beta1*q);
}
double rDs_FC () {
  return(beta1*(1-q));
}

/* CARDINALITY Functions */

void net() {

  /*  PLACE  */       /* Active HSP          Standby HSP  */
  place("hsp2");    /* no fault           no fault    */
  init("hsp2",1);
  place("Ds");       /* fault not covered, reboot    no fault    */
  place("FC");       /* reboot failed, repair    no faul    */
  /*  TRANSITION  */
  ratefun("hsp2_Ds",rHsp2_Ds);
  ratefun("Ds_hsp2",rDs_hsp2);
  ratefun("Ds_FC",rDs_FC);
  rateval("FC_hsp2",rrPo);
  /*  ARC  */
  iarc("hsp2_Ds","hsp2");
  oarc("hsp2_Ds","Ds");
  iarc("Ds_hsp2","Ds");
  oarc("Ds_hsp2","hsp2");
  iarc("Ds_FC","Ds");
  oarc("Ds_FC","FC");
  iarc("FC_hsp2","FC");
  oarc("FC_hsp2","hsp2");
}

void assert() {

}

void ac_init() {

}

void ac_reach() {

}

void ac_final() {
  solve(INFINITY);
```

```
  pr_expected("dtime",dtime);

  pr_message("\n------------------\n");
  pr_message("\n(This is what actually used ");
  pr_message("and fed into upper level.)\n");
  pr_message("\nequivalents for the subsystem:\n");
  hold_cond(cond,times);
  pr_value("MTTFeq",times[0]);
  pr_value("MTTReq",times[1]);
  pr_value("ofm: ", (1-expected(dtime)/(8766*60)) / times[0] *8766);
  pr_message("\nequivalent DPM computed from MTTFeq and MTTReq:\n");
  pr_value("DPMeq", times[1]/(times[1]+times[0])*8766*60);

  pr_mc_info();
  pr_std_average();
}
```

10.16.5 SPNP File for System level SPN model — *gsb.c*

```
#include <stdio.h>
#include "user.h"

/* global variables */

/* all the rates are in the format 1.0/MTTF or 1.0/MTTR (unit: 1/Hour) */

/* global variables for other parts of the system */

/* number of power supply+fan modules */
int numPsf=3;
int numPsfUpMin=2;    /* minimum number to make system up */

/* number of NSPs on bus A and B */
int numNspA=4;
int numNspB=4;
int numNsp=8;
int numNspFcMin=4;    /* minimum number to make system full capacity */
int numNspUpMin=1;

/* number of DSPs on bus A and B */
int numDspA=2;
int numDspB=2;
int numDsp=4;
int numDspFcMin=2;    /* minimum number to make system full capacity */
int numDspUpMin=1;

/* failure rate of HSP */
double frHsp;

/* failure rate of backplane */
double frBp=1.0/6000000.0;

/* failure rate of power supply+fan */
double frPsf=1.0/80000.0;

/* failure rate of bus */
double frBus=1.0/300000.0;

/* failure rate of NSP */
```

```
double frNsp=1.0/100000.0;  /* change later */

/* failure rate of DSP */
double frDsp=1.0/100000.0;  /* change later */

/* repair rate of backplane */
double rrBp=1.0/6.0;

/* repair rate of Ps+Fn, Bus, NSP, DSP  */
/*   in lost redudency/partial outage/total outage */
double rrLr=1.0/12.0;
double rrPo=1.0/8.0;
double rrTo=1.0/4.0;

double rrHsp,rrNsp,rrDsp;

/* global variables for computing capacity */

double dnNsp, dnDsp, upNsp, upDsp;
double capHsp, capBp, capPsf, capBus;
double capNsp, capDsp, cap;    /* capacity */



/* prototype reward functions */
double uodpm();
double tpodpm();
double ofm();
double tsodpm();
double tsodpmHsp();
double tsodpmBp();
double tsodpmPsf();
double tsodpmBus();
double tsodpmNsp();
double tsodpmDsp();
void calCap();    /* calculate capacity */
void calVar();
int condTo();
int condTso();


/* prototype guard functions */

/* prototype rate functions */
double rrfPsf();
double rrfBus();

/* prototype cardinality functions */



void options() {
  iopt(IOP_PR_RGRAPH,VAL_NO) ;
  iopt(IOP_PR_MC,VAL_NO) ;
  iopt(IOP_PR_DERMC,VAL_NO) ;
  iopt(IOP_PR_PROB,VAL_NO) ;
  iopt(IOP_PR_PROBDTMC,VAL_NO) ;
  iopt(IOP_PR_DOT,VAL_NO) ;
  iopt(IOP_PR_MERG_MARK,VAL_YES) ;
  iopt(IOP_PR_FULL_MARK,VAL_NO) ;
  iopt(IOP_USENAME,VAL_NO) ;
  iopt(IOP_DEBUG,VAL_NO) ;
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC) ;
```

```
  iopt(IOP_PR_RSET,VAL_NO) ;
  iopt(IOP_PR_MC_ORDER,VAL_FROMTO) ;
  /* NUMERICAL SOLUTION chosen */
  iopt(IOP_SENSITIVITY,VAL_NO) ;
  iopt(IOP_MC,VAL_CTMC) ;
  iopt(IOP_SSMETHOD,VAL_SSSOR) ;
  iopt(IOP_TSMETHOD,VAL_FOXUNIF) ;
  iopt(IOP_ITERATIONS,2000) ;
  fopt(FOP_PRECISION,0.000001) ;
  fopt(FOP_ABS_RET_M0,0.0) ;
  iopt(IOP_CUMULATIVE,VAL_NO) ;
  iopt(IOP_SSDETECT,VAL_YES) ;
  iopt(IOP_OK_ABSMARK,VAL_NO) ;
  iopt(IOP_OK_VANLOOP,VAL_NO) ;
  iopt(IOP_OK_TRANS_M0,VAL_YES) ;
  iopt(IOP_OK_VAN_M0,VAL_YES) ;
  iopt(IOP_ELIMINATION,VAL_REDONTHEFLY) ;

  /* some computation */
  calVar();

}

void calVar() {
  double eq;
  FILE *pp;
  char command[120]="run gsbhsp";
  char cmdvsc[120];
  system(command);
  if ((pp=popen("grep MTTFeq gsbhsp.out | awk '{print $4}'","r"))==NULL) {
    perror("popen");
    exit(1);
  }
  fscanf(pp,"%14lf",&eq);
  printf("%.12f\n",eq);
  fclose(pp);
  frHsp = 1/eq;
  if ((pp=popen("grep MTTReq gsbhsp.out | awk '{print $4}'","r"))==NULL) {
    perror("popen");
    exit(1);
  }
  fscanf(pp,"%14lf",&eq);
  printf("%.12f\n",eq);
  fclose(pp);
  rrHsp = 1/eq;

  strcpy(cmdvsc,"run gsbvsc1");
  system(cmdvsc);
  if ((pp=popen("grep MTTFeq gsbvsc1.out | awk '{print $4}'","r"))==NULL) {
    perror("popen");
    exit(1);
  }
  fscanf(pp,"%14lf",&eq);
  printf("%.12f\n",eq);
  fclose(pp);
  frNsp = 1/eq;
  frDsp = 1/eq;
  if ((pp=popen("grep MTTReq gsbvsc1.out | awk '{print $4}'","r"))==NULL) {
    perror("popen");
    exit(1);
  }
  fscanf(pp,"%14lf",&eq);
```

```
  printf("%.12f\n",eq);
  fclose(pp);
  rrNsp = 1/eq;
  rrDsp = 1/eq;
}

/* REWARD Functions */

/* Unweighted Outage Downtime Performance Measurement */

double uodpm() {
  double dt;
  calCap();
  if ( cap<0.99999 )  { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

int condUo() {
  calCap();
  if ( cap<0.99999 )  { return 1; }
  else { return 0; }
}

double timesUo[2];

/* Partial Outage Downtime Performance Measurement */

double tpodpm() {
  double dt;
  calCap();
  dt = 1-cap;
  dt *= 8766*60;
  return dt;
}

double tpodpmNsp() {
  double dt;
  calCap();
  dt = 1-capNsp;
  dt *= 8766*60;
  return dt;
}

double tpodpmDsp() {
  double dt;
  calCap();
  dt = 1-capDsp;
  dt *= 8766*60;
  return dt;
}

double tpodpmBus() {
  double dt;
  calCap();
  dt = 1-capBus;
  dt *= 8766*60;
  return dt;
```

```
}

/* Outage Frequency Measurement */

double ofm() {}

/* Total System Outage DPM */

double tsodpm() {
  double dt;
  calCap();
  if ( cap<0.00001 )  { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

int condTso() {
  calCap();
  if ( cap<0.00001 )  { return 1; }
  else { return 0; }
}

double timesTso[2];

/* total system outage caused by HSP */

double tsodpmHsp() {
  double dt;
  calCap();
  if ( capHsp<0.00001 )  { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

double tsodpmBp() {
  double dt;
  calCap();
  if ( capBp<0.00001 ) { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

double tsodpmPsf() {
  double dt;
  calCap();
  if ( capPsf<0.00001 ) { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

double tsodpmBus() {
  double dt;
  calCap();
  if ( capBus<0.00001 ) { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
```

```
  return dt;
}

double tsodpmNsp() {
  double dt;
  calCap();
  if ( capNsp<0.00001 )  { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

double tsodpmDsp() {
  double dt;
  calCap();
  if ( capDsp<0.00001 )  { dt=1; }
  else { dt=0; }
  dt *= 8766*60;
  return dt;
}

void calCap() {
  cap=1;
  if ( mark("hspUp")==0 )  { capHsp=0; cap=0; }
  else { capHsp=1; }
  if ( mark("bpUp")==0 )  { capBp=0; cap=0; }
  else { capBp=1; }
  if ( mark("psfUp")<numPsfUpMin ) { capPsf=0; cap=0; }
  else { capPsf=1; }
//if ( mark("busAUp")==0 && mark("busBUp")==0 ) { capBus=0; cap=0; }
//else { capBus=1; }
  capBus = (mark("busAUp") + mark("busBUp"))/2.0;
  if (capBus < 0.00001) cap=0;

  upNsp = mark("nspAUp")*mark("busAUp") + mark("nspBUp")*mark("busBUp");
  upDsp = mark("dspAUp")*mark("busAUp") + mark("dspBUp")*mark("busBUp");
  if ( upNsp<numNspFcMin-0.00001 )  { capNsp = (double)upNsp/numNspFcMin; }
  else { capNsp=1; }
  if ( upDsp<numDspFcMin-0.00001 )  { capDsp = (double)upDsp/numDspFcMin; }
  else { capDsp=1; }
  if (cap>0.99999)  { cap = ( capNsp < capDsp ) ?  capNsp : capDsp; }
}



/* GUARD Functions */

/* RATE Functions */

double rrfPsf() {
  calCap();
  if ( cap<0.00001 ) { return rrTo; }
  else { return rrLr; }
}

double rrfBus() {
  calCap();
  if ( cap<0.00001 ) { return rrTo; }
```

```
  else { return rrPo; }
}

/* CARDINALITY Functions */


void net() {

  /* PLACE */

  place("hspUp");
  init("hspUp",1);
  place("hspDn");
  place("bpUp");
  init("bpUp",1);
  place("bpDn");
  place("psfUp");
  init("psfUp",numPsf);
  place("psfDn");
  place("busAUp");
  init("busAUp",1);
  place("busADn");
  place("busBUp");
  init("busBUp",1);
  place("busBDn");
  place("nspAUp");
  init("nspAUp",numNspA);
  place("nspADn");
  place("nspBUp");
  init("nspBUp",numNspB);
  place("nspBDn");
  place("dspAUp");
  init("dspAUp",numDspA);
  place("dspADn");
  place("dspBUp");
  init("dspBUp",numDspB);
  place("dspBDn");

  /* TRANSITION */

  ratedep("hspFail", frHsp, "hspUp");
  ratedep("hspRepr", rrHsp, "hspDn");
  ratedep("bpFail", frBp, "bpUp");
  ratedep("bpRepr", rrBp, "bpDn");
  ratedep("psfFail", frPsf, "psfUp");
  ratefun("psfRepr", rrfPsf);
  ratedep("busAFail", frBus, "busAUp");
  ratefun("busARepr", rrfBus);
  ratedep("busBFail", frBus, "busBUp");
  ratefun("busBRepr", rrfBus);
  ratedep("nspAFail", frNsp, "nspAUp");
  ratedep("nspARepr", rrNsp, "nspADn");
  ratedep("nspBFail", frNsp, "nspBUp");
  ratedep("nspBRepr", rrNsp, "nspBDn");
  ratedep("dspAFail", frDsp, "dspAUp");
  ratedep("dspARepr", rrDsp, "dspADn");
  ratedep("dspBFail", frDsp, "dspBUp");
  ratedep("dspBRepr", rrDsp, "dspBDn");

  /* ARC */

  iarc("hspFail", "hspUp");
```

```
  oarc("hspFail", "hspDn");
  iarc("hspRepr", "hspDn");
  oarc("hspRepr", "hspUp");
  iarc("bpFail", "bpUp");
  oarc("bpFail", "bpDn");
  iarc("bpRepr", "bpDn");
  oarc("bpRepr", "bpUp");
  iarc("psfFail", "psfUp");
  oarc("psfFail", "psfDn");
  iarc("psfRepr", "psfDn");
  oarc("psfRepr", "psfUp");
  iarc("busAFail", "busAUp");
  oarc("busAFail", "busADn");
  iarc("busARepr", "busADn");
  oarc("busARepr", "busAUp");
  iarc("busBFail", "busBUp");
  oarc("busBFail", "busBDn");
  iarc("busBRepr", "busBDn");
  oarc("busBRepr", "busBUp");
  iarc("nspAFail", "nspAUp");
  oarc("nspAFail", "nspADn");
  iarc("nspARepr", "nspADn");
  oarc("nspARepr", "nspAUp");
  iarc("nspBFail", "nspBUp");
  oarc("nspBFail", "nspBDn");
  iarc("nspBRepr", "nspBDn");
  oarc("nspBRepr", "nspBUp");
  iarc("dspAFail", "dspAUp");
  oarc("dspAFail", "dspADn");
  iarc("dspARepr", "dspADn");
  oarc("dspARepr", "dspAUp");
  iarc("dspBFail", "dspBUp");
  oarc("dspBFail", "dspBDn");
  iarc("dspBRepr", "dspBDn");
  oarc("dspBRepr", "dspBUp");

}

void assert() {

}

void ac_init() {

}

void ac_reach() {

}

void ac_final() {
/*
  double t;
  for (t=0; t≤50000; t+=30000)  {
    solve(t);
    pr_expected("dtime",dtime);
    pr_expected("dtimeFC",dtimeFC);
    pr_expected("dtimeF",dtimeF);
    pr_expected("tdpm", tdpm);
  }
*/
  solve(INFINITY);
```

```
  pr_expected("uodpm", uodpm);

  pr_message("\n------------------\n");
  pr_expected("tpodpm", tpodpm);
  pr_expected("tpodpmNsp", tpodpmNsp);
  pr_expected("tpodpmDsp", tpodpmDsp);
  pr_expected("tpodpmBus", tpodpmBus);
  pr_value("podpm", expected(tpodpm)−expected(tsodpm));
  pr_value("podpmNsp", expected(tpodpmNsp)−expected(tsodpmNsp));
  pr_value("podpmDsp", expected(tpodpmDsp)−expected(tsodpmDsp));
  pr_value("podpmBus", expected(tpodpmBus)−expected(tsodpmBus));

  pr_message("\n------------------\n");
  pr_expected("tsodpm", tsodpm);
  pr_expected("tsodpmHsp", tsodpmHsp);
  pr_expected("tsodpmBp", tsodpmBp);
  pr_expected("tsodpmPsf", tsodpmPsf);
  pr_expected("tsodpmBus", tsodpmBus);
  pr_expected("tsodpmNsp", tsodpmNsp);
  pr_expected("tsodpmDsp", tsodpmDsp);

  pr_message("\n------------------\n");
  pr_message("\nequivalents for the UO (unweighted) of the system:\n");
  hold_cond(condUo,timesUo);
  pr_value("UO-MTTFeq",timesUo[0]);
  pr_value("UO-MTTReq",timesUo[1]);
  pr_value("UO-OFM: ", (1−expected(uodpm)/(8766∗60)) / timesUo[0] ∗8766);
  pr_message("\nequivalent DPM computed from MTTFeq and MTTReq:\n");
  pr_value("UODPMeq", timesUo[1]/(timesUo[1]+timesUo[0])∗8766∗60);

  pr_message("\n------------------\n");
  pr_message("\nequivalents for the TSO of the system:\n");
  hold_cond(condTso,timesTso);
  pr_value("TSO-MTTFeq",timesTso[0]);
  pr_value("TSO-MTTReq",timesTso[1]);
  pr_value("TSOFM: ", (1−expected(tsodpm)/(8766∗60)) / timesTso[0] ∗8766);
  pr_message("\nequivalent DPM computed from MTTFeq and MTTReq:\n");
  pr_value("TSODPMeq", timesTso[1]/(timesTso[1]+timesTso[0])∗8766∗60);

  pr_mc_info();
  pr_std_average();
}
```

10.17 Analysis of Phased-Mission Systems (PMS) with DSPN

10.17.1 Source

I. Mura, A. Bondavalli, X. Zang, and K.S. Trivedi. ”Dependability modeling and evaluation of phased mission systems: a DSPN approach”. In *Proc. IFIP International Conference on Dependable Computing for Critical Applications (DCCA-7)*, pages 299–318, San Jose, California, January 1999.

10.17.2 Description

Many systems used in the control and management of critical activities perform a series of tasks that are accomplished in sequence. The operational life of these systems consists of consecutive, non-overlapping time periods, called *phases*, during which the system configuration, success criteria, and component behavior may vary from phase to phase. These variations may be due to different tasks in each phase, or different conditions of the environment, as well as different dependability requirements and failure scenarios. In order to accomplish their missions, systems need to change their configuration over time to adopt suitable one in accordance with the performance and dependability requirements of current phase. Many practical systems are actually phased mission systems, e.g., the voyage of an aircraft can be divided into several phases, such as take-off, cruise and landing, each with completely different reliability requirements and behaviors.

Compared with single-phased systems, the reliability analysis of PMS is much more complex because of the dependence across phases. The dynamic structure and configuration of the PMS usually requires a distinct model for each phase, which also increases the complexity of modeling and analysis. Here, SPNP is used to analyze the PMS.

Fig. 10.23 shows a general model scheme of a PMS. Here we assume that the only deterministic activities in PMS are the phase changes. At a high abstraction level, the model of a PMS is composed of two logically separate subnets:

- system subnet:
  This subnet, called system net (SN), represents components in the system and their interactions, which evolve according to the events that modify system states. As seen in the Figure 10.23, this subnet is a pure SRN that contains only exponentially distributed and immediate transitions.

- phase subnet:
  This subnet describes the phase changes. It is represented by a DSPN, and is called the phase net (PhN). The PhN contains all the deterministic transitions of the overall DSPN model and may contain immediate transitions as well.

10.17.3 SPNP File — *pms.c*

```
#include "user.h"
#include <string.h>

#define MAXPHASE 30

static struct initpv {
  double prob;
  int nToken[5];
} initPV[10];
```

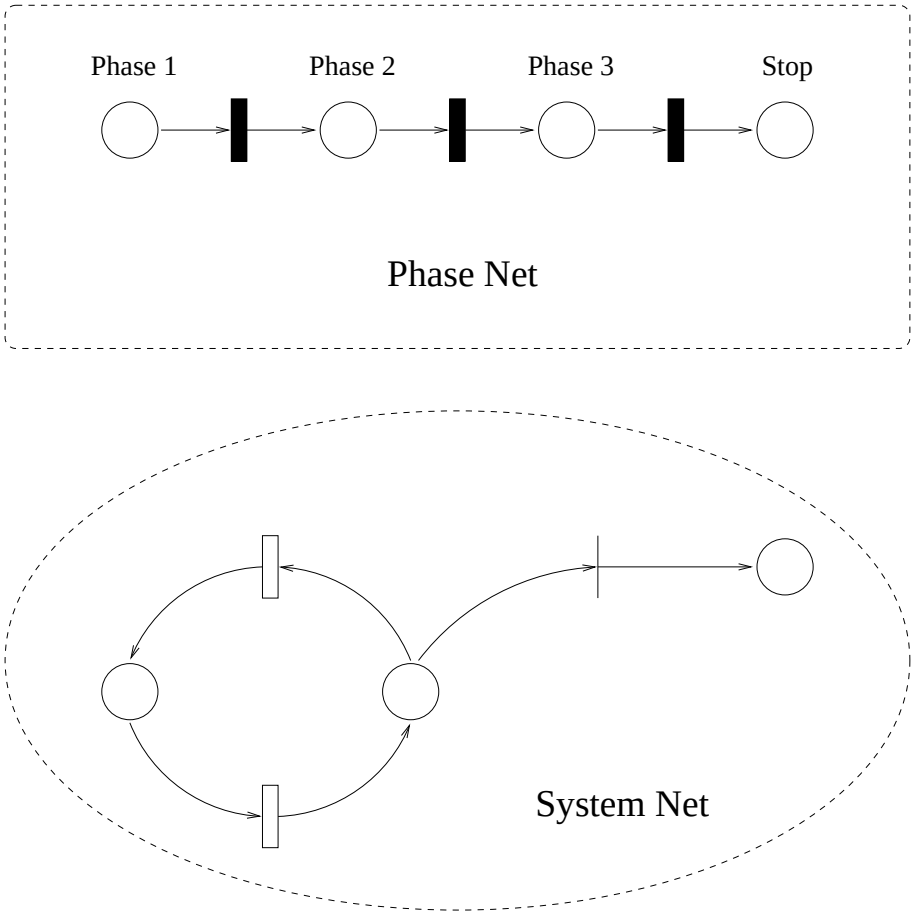


Figure 10.23: DSPN model of a PMS

```
static double pTime[MAXPHASE], failrate[MAXPHASE];
static double reprate[MAXPHASE], coverage[MAXPHASE];

static int upreq[MAXPHASE], downreq[MAXPHASE], numvp[MAXPHASE];

static int TNUM = 4, nPlace = 5, nState = 0, nPhase = 0, ntPhase = 1;

static char sPlace[5][10] = {"P_LPH", "P_UP", "P_DOWN", "P_SPARE", "P_FAIL"};

static int UPREQ, DOWNREQ;

static double phaseTime, NUMVP, FAILRATE, REPRATE, C;

void LoadP(FILE *fp, void *data, int type)
{
  int *iData, idData, i;
  double *fData, fdData;
  char sstr[100], *ptr;

  if (type)
    fData = (double *)data;
  else
    iData = (int *)data;

  while(fgets(sstr, 100, fp)) {
    if (!strncmp(sstr, "default:", 8)) {
      ptr = strchr(sstr, ':');
      if (type)
    sscanf(ptr + 1, "%lf", &fdData);
      else
    sscanf(ptr + 1, "%d", &idData);
      for (i = 0; i < ntPhase; i ++)
    if (type)
      fData[i] = fdData;
    else
      iData[i] = idData;
      continue;
    }
    if (!strncmp(sstr, "end", 3))
      break;
    sscanf(sstr, "%d", &i);
    if ((i ≥ 0) && (i < ntPhase)) {
      ptr = strchr(sstr, ':');
       if (type)
    sscanf(ptr + 1, "%lf", &fData[i]);
      else
    sscanf(ptr + 1, "%d", &iData[i]);
    }
  }
}

void LoadConfig(char *sModelName)
{
  FILE *fp;
  char sstr[100], *ptr;

  sprintf(sstr, "%s.cfg", sModelName);
  if ((fp = fopen(sstr, "r")) == NULL) {
    perror("Can not Open the Config File");
    exit(1);
  }
```

```
  while (fgets(sstr, 100, fp)) {
   if (*sstr == '#' || *sstr == '\n')
    continue;

   if (!strncmp(sstr, "Number of Phase:", 16)) {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%d", &ntPhase);
    continue;
   }

   if (!strncmp(sstr, "Number of CPU:", 14)) {
    ptr = strchr(sstr, ':');
    sscanf(ptr + 1, "%d", &TNUM);
    continue;
   }

   if (!strncmp(sstr, "Failure Rate:", 13)) {
    LoadP(fp, (void *)failrate, 1);
    continue;
   }

   if(!strncmp(sstr, "Repair Rate:", 12)) {
    LoadP(fp, (void *)reprate, 1);
    continue;
   }

   if(!strncmp(sstr, "Number of Step:", 15)) {
    LoadP(fp, (void *)numvp, 0);
    continue;
   }

   if(!strncmp(sstr, "Coverage:", 9)) {
    LoadP(fp, (void *)coverage, 1);
    continue;
   }

   if(!strncmp(sstr, "Phase Time:", 11)) {
    LoadP(fp, (void *)pTime, 1);
    continue;
   }

   if(!strncmp(sstr, "CPU Requirement:", 15)) {
    LoadP(fp, (void *)upreq, 0);
    continue;
   }

   if(!strncmp(sstr, "CPU Down Cause Fail:", 20)) {
    LoadP(fp, (void *)downreq, 0);
    continue;
   }
  }
  fclose(fp);
}

void LoadState(char *sModelName)
{
  FILE *fp;
  char sstr[100], *ptr, nstr[10];
  int i, j, num, np;
```

```
  sprintf(sstr, "%s.rg", sModelName);
  if ((fp = fopen(sstr, "r")) == NULL) {
    perror("Can not Open the Output File of Last Phase");
    exit(1);
  }

  while (fgets(sstr, 100, fp)) {
    if (!strncmp(sstr, "_ntanmark", 8)) {
      sscanf(sstr + 11, "%d", &num);
      nState += num;
      continue;
    }

    if (!strncmp(sstr, "_nabsmark", 8)) {
      sscanf(sstr + 11, "%d", &num);
      nState += num;
      continue;
    }

    if (!strncmp(sstr, "_nvanmark", 8)) {
      sscanf(sstr + 11, "%d", &num);
      nState += num;
      continue;
    }

    if (!strncmp(sstr, "_reachset", 8)) {
      num = nState + 1;
      for (i = 0; i < num; i ++) {
    fgets(sstr, 100, fp);
    if (*sstr == '#')
      continue;
    sscanf(sstr, "%d", &np);
    ptr = strchr(sstr, '_');
    if (*(++ptr) == 'v') {
      nState --;
      continue;
    }

    ptr ++;
    for (j = 0; j < nPlace; j ++) {
      sscanf(ptr, "%s", nstr);
      if (*nstr == ':')
        initPV[np].nToken[j] = 0;
      else
        sscanf(nstr, "%d", &initPV[np].nToken[j]);
      ptr = strstr(ptr, nstr);
      ptr += strlen(nstr);
    }
      }
      break;
    }
  }
  fclose(fp);
}

void LoadProb(char *sModelName)
{
  FILE *fp;
  char sstr[100], *ptr, nstr[40], c;
  int i, np;
  double ttime;
```

```
  sprintf(sstr, "%s.prb", sModelName);
  if ((fp = fopen(sstr, "r")) == NULL) {
    perror("Can not Open the Output File of Last Phase");
    exit(1);
  }

  while (fgets(sstr, 100, fp)) {
    if (!strncmp(sstr, "_time", 5)) {
      sscanf(sstr + 8, "%lf", &ttime);
      if (ttime ≠ pTime[nPhase − 1])
    continue;
      fgets(sstr, 100, fp);
      np = 0;
      while (fgets(sstr, 100, fp)) {
    ptr = sstr;
    while (1) {
      sscanf(ptr, "%s", nstr);
      if (!strlen(nstr))
        break;
      sscanf(nstr, "%d%c%lf", &i, &c, &initPV[np].prob);
      np ++;
      if (np ≥ nState)
        break;
      ptr = strstr(ptr, nstr);
      ptr += strlen(nstr);
      if (ptr − sstr ≥ strlen(sstr))
        break;
      else if (!strstr(ptr, ":"))
        break;
    }
    if (np ≥ nState)
        break;
      }
      break;
    }
  }
  fclose(fp);
}

void LoadPhaseParameter(char ∗sPhase)
{
  char ∗ptr;
  int n;

  if ((ptr = strchr(modelname, '_')) == NULL) {
    printf("Wrong Model Name.\n");
    exit(1);
  }

  sscanf(ptr + 1, "%d", &n);
  nPhase = n;
  strncpy(sPhase, modelname, ptr − modelname);
  ∗(sPhase + (ptr − modelname)) = '\0';

  LoadConfig(sPhase);

  phaseTime = pTime[n];
  UPREQ = upreq[n];
  DOWNREQ = downreq[n];
  FAILRATE = failrate[n];
  REPRATE = reprate[n];
```

```
  C = coverage[n];
  NUMVP = (double)numvp[n];

  if (n) {
    sprintf(sPhase + (ptr − modelname), "_%d", n − 1);
  }
}

void options() {
  char LastPhase[20];

  iopt(IOP_TSMETHOD,VAL_TSUNIF);
  iopt(IOP_PR_FULL_MARK,VAL_YES);
/* iopt(IOP_ELIMINATION, VAL_REDAFTERRG); */
  iopt(IOP_OK_TRANS_M0, VAL_YES);
  iopt(IOP_OK_VAN_M0, VAL_YES);
  iopt(IOP_OK_VANLOOP, VAL_YES);
  iopt(IOP_PR_MARK_ORDER,VAL_CANONIC);
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_MC,VAL_YES);
  iopt(IOP_PR_PROB,VAL_YES);
  iopt(IOP_MC,VAL_CTMC);
  iopt(IOP_PR_RSET,VAL_YES);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  iopt(IOP_ITERATIONS,20000);
  fopt(FOP_ABS_RET_M0,0.0);
  fopt(FOP_PRECISION,0.00000001);

  LoadPhaseParameter(LastPhase);
  if(nPhase) {
    LoadState(LastPhase);
    LoadProb(LastPhase);
  }

}

int GFRec()
{
  if (mark("P_UP") < UPREQ)
    return 1;
  else
    return 0;
}

int GFShutdown()
{
  if (mark("P_UP") > UPREQ)
    return 1;
  else
    return 0;
}

int GFSysfail()
{
  if (mark("P_DOWN") ≥ DOWNREQ)
    return 1;
  else
    return 0;
}

int GFSysfailu()
{
```

```
 if ((mark("P_DOWN") ≥ DOWNREQ) && (mark("P_UP") > 0))
   return 1;
 else
   return 0;
}

int VFUp()
{
 return mark("P_UP");
}

int VFDown()
{
 return mark("P_DOWN");
}

double RFSysDown()
{
 if (mark("P_FAIL"))
   return 0.0;
 else
   return 1.0;
}

void SetInitProb(int nph)
{
 int i, j;
 char tstr[20];

 if (nph) {
   for (i = 0; i < nState; i ++) {
     if (initPV[i].prob < 0.00000001)
   continue;
     sprintf(tstr, "IT_IP%d", i);
     imm(tstr);
     probval(tstr, initPV[i].prob);
     miarc(tstr, "P_LPH", TNUM);
     for (j = 0; j < nPlace; j ++)
   if (initPV[i].nToken[j])
     moarc(tstr, sPlace[j], initPV[i].nToken[j]);
   }
 } else {
   imm("IT_IP");
   probval("IT_IP", 1.0);
   miarc("IT_IP", "P_LPH", TNUM);
   moarc("IT_IP", sPlace[1], TNUM);
 }
}

void net()
{
 place("P_LPH"); init("P_LPH", TNUM);

 SetInitProb(nPhase);

 place("P_UP");
 place("P_DOWN");
 place("P_SPARE");
 place("P_FAIL");

 ratedep("T_FAIL", FAILRATE, "P_UP");
 iarc("T_FAIL", "P_UP");
```

```
  oarc("T_FAIL", "P_DOWN");

  ratedep("T_REPAIR", REPRATE, "P_DOWN");
  iarc("T_REPAIR", "P_DOWN");
  oarc("T_REPAIR", "P_UP");

  imm("IT_RECS");
  probval("IT_RECS", C);
  guard("IT_RECS", GFRec);
  priority("IT_RECS", 10);
  iarc("IT_RECS", "P_SPARE");
  oarc("IT_RECS", "P_UP");

  imm("IT_RECF");
  probval("IT_RECF", 1 − C);
  guard("IT_RECF", GFRec);
  priority("IT_RECF", 10);
  iarc("IT_RECF", "P_SPARE");
  oarc("IT_RECF", "P_DOWN");

  imm("IT_SHUTDOWN");
  probval("IT_SHUTDOWN", 1);
  guard("IT_SHUTDOWN", GFShutdown);
  priority("IT_SHUTDOWN", 10);
  iarc("IT_SHUTDOWN", "P_UP");
  oarc("IT_SHUTDOWN", "P_SPARE");

  imm("IT_SYSFAIL1");
  probval("IT_SYSFAIL1", 1);
  guard("IT_SYSFAIL1", GFSysfailu);
  priority("IT_SYSFAIL1", 100);
  viarc("IT_SYSFAIL1", "P_UP", VFUp);
  voarc("IT_SYSFAIL1", "P_FAIL", VFUp);

  imm("IT_SYSFAIL2");
  probval("IT_SYSFAIL2", 1);
  guard("IT_SYSFAIL2", GFSysfail);
  priority("IT_SYSFAIL2", 10);
  viarc("IT_SYSFAIL2", "P_DOWN", VFDown);
  voarc("IT_SYSFAIL2", "P_FAIL", VFDown);
}

int assert() {
  return(RES_NOERR);
}

void ac_init() {
  pr_net_info(); /* information on the net structure */
}

void ac_reach() {
  pr_rg_info(); /* information on the reachability graph */
}

void
ac_final() {
  double ttime, tstep = phaseTime / NUMVP;

  for (ttime = 0; ttime ≤ phaseTime; ttime += tstep) {
    solve(ttime);
    pr_expected("System Fail", RFSysDown);
  }
```

```
  solve(phaseTime);
  pr_mc_info();
  pr_std_average(); /* default measures */
}
```

10.17.4 Configuration File — *pms.cfg*

```
# set number of phase
Number of Phase: 7

# set total number of CPU
Number of CPU: 4

# set failure rate of CPU, can be set as default or specific phase
Failure Rate:
default: 0.00001
0: 0.001
2: 0.0001
4: 0.0001
end

# set repair rate of CPU, can be set as default or specific phase
Repair Rate:
default: 0.001
end

# set coverage for reconfiguration, can be set as default or specific phase
Coverage:
default: 0.9999
end

# set number of point to obtain result in each phase
Number of Step:
default: 10
0: 8
2: 7
4: 7
end

# set period of each phase
Phase Time:
default: 672
0: 48
1: 17520
3: 26280
end

# set CPU requirement for system working in each phase
CPU Requirement:
default: 3
1: 2
3: 2
end
```

```
# set the number of CPU down will cause system down.
CPU Down Cause Fail:
default: 2
1: 4
3: 4
end
```

10.17.5 Shell File — *t.csh*

This shell file is used to run the example. First use SPNP package to compile the cspl file *pms.c*, then run *t.csh* to get the result.

```
#!/bin/csh
set Model = "pms"
foreach nPhase (0 1 2 3 4)
  set ModelName = "$Model""_""$nPhase"
  pms.spn $ModelName
end
```

10.18 Extensions to SPNP

In the use of SPNP to solve real world problems, we often generate a system model that is too big to be solved by SPNP. A solution, which has been used extensively in the previous examples, is dividing the model into several SPNs and solving it by iteratively execution of these SPNs. In the following, two techniques we have been developed for this task will be introduced.

10.18.1 Fixed point iteration

Suppose we have two submodels M_1 and M_2 which are all SPNs. Some parameters of M_2 depend on the statistical behavior of M_1, and M_1 also has some parameters depend on M_2. To solve this kind of interconnected models, fixed point iteration have to be used:

1. Set error bound e as a small real number.
2. Initialze the unknown parameters of M_1, $(P_{11}, P_{12}, \ldots, P_{1m})$, to some reasonable random values $(P_{1i}^{(0)})$.
3. Execute M_1, compute the parameters $(P_{21}^{(0)}, P_{22}^{(0)}, \ldots, P_{2n}^{(0)})$ required by M_2.
4. Set $k := 1$.

5. Execute M_2 with the parameters obtained from last step, and compute the parameters $(P_{11}^{(k)}, P_{12}^{(k)}, \ldots, P_{1m}^{(k)})$.

6. Execute M_1 with the new set of parameters, and compute $(P_{21}^{(k)}, P_{22}^{(k)}, \ldots, P_{2n}^{(k)})$.

7. If $\sum |P_i^(k) - P_i^(k-1)| < e$, then stop, else set $k := k + 1$ and goto step 5.

Under a very general condition, we can prove that the solution always exists, but the uniqueness of the solution is not guaranteed (See [4]). However, the meaning of practical problems often has uniqueness itself, so the justification is enough for the practical use of fixed point iterations.

10.18.2 Initial probability reload

The function

```
void loadprob(char *fname);
```

is usually used before a transient anaylsis to set initial probabilities to the current Petri nets from a Petri net that shares (1) the same place names; and (2) the same state space. Input parameter (char *fname) specifies the file name (no extension needed) of the reachbility graph file (.rg) and the prabability file (.prb). Both files must exist in the same directory to make a successful call. The function is suggested to be called in the cspl file before the function, `solve()`, which invokes a transient analysis. The function is useful in the (transient) analysis of phase-mission systems.

(A future version will break the limitation that both Petri nets have the same state space and place names, It is supposed to able to read user-specified rules of mapping from the state space of input Petri net to the current Petri net.)

/*

Module:
loadprob.c

Author:
Yonghuan Cao (ycao@ee.duek.edu)

Changes:
10/13/1999 (initial version)

Functions:

static (internal)

void read_rg_head(void);
void next_marking(void);

```
   void read_prb_head(void);
   double next_probability(void);

   global (can be called from outside)

   void loadprob(char *sfName);
*/

#include     "sysinclude.h"
#include     "port.h"
#include     "const.h"
#include     "type.h"
#include     "var.h"
#include     "options.h"
#include     "reach.h"
#include     "cspl.h"
#include     "rdc.h"
#include     "utility.h"

#include   <string.h>
#include   <math.h>

#define MAXPLACE    64
#define MAXPLNAME    32
#define MAXLINE       (MAXPLACE*8)

static FILE *fpRG, *fpPrb;

static int nplace = 0;
static int nstates = 0;
static double time0 = 0.00;
static char splace[MAXPLACE][MAXPLNAME];
static int idx[MAXPLACE];
static int marking[MAXPLACE];
static char sline[MAXLINE];

void read_rg_head() {

  int i;
  char *p;

  /* read _nplace */
  while(!feof(fpRG)) {
    fgets(sline, MAXLINE, fpRG);
    if ((p = strstr(sline, "_nplace"))) {
      break;
    }
  }

  if (feof(fpRG)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "read_prb_head reaches EOF unexpected.");
  }

  strtok(p, "=; "); /* skip "_nplace" */
  p = strtok((char*)NULL, "=; ");
  nplace = atoi(p);
  printf("nplace = %d\n", nplace);

  /* read place names */
  while(!feof(fpRG)) {
    fgets(sline, MAXLINE, fpRG);
```

```
    if ((p = strstr(sline, "_places"))) {
      break;
    }
  }

  if (feof(fpRG)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "read_prb_head reaches EOF unexpected.");
  }

  for(i = 0; i < nplace; i ++) {
    fgets(sline, MAXLINE, fpRG);
    strtok(sline, ":; "); /* skip "i:" */
    p = strtok((char *)NULL, ":; ");
    if ((idx[i] = findplace(p)) == RES_ERROR) {
      LogMsg(MSG_EXIT, "'load_init_prob' could not open .rg file.");
    }
    strcpy(splace[i], p);
    printf("%d: %s\n", i, p);
  }

  /* go to the start of markings */
  p = (char *)NULL;
  while(!feof(fpRG)) {
    fgets(sline, MAXLINE, fpRG);
    if ((p = strstr(sline, "_reachset"))) {
      break;
    }
  }

  if (feof(fpRG)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "read_prb_head reaches EOF unexpected.");
  }

  /* skip the line w/ place names */
  fgets(sline, MAXLINE, fpRG);

  /* now we are ready to read markings */
}

void next_marking() {

  int i;
  char *p;

  /* end-of-file reached */
  if (feof(fpRG)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "next marking: reaches EOF unexpected.");
  }

  fgets(sline, MAXLINE, fpRG);
  strtok(sline, " "); /* skip the 1st token "i_t" */
  for(i = 0; i < nplace; i ++) {
    p = strtok((char *)NULL, " ");
    if (p[0] == ':') {
      marking[i] = 0;
```

```
    } else {
      marking[i] = atoi(p);
    }
  }
}

void read_prb_head() {

  char *p;

  /* read _nstates */
  while(!feof(fpPrb)) {
    fgets(sline, MAXLINE, fpPrb);
    if ((p = strstr(sline, "_nstates"))) {
      break;
    }
  }

  if (feof(fpPrb)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "read_prb_head reaches EOF unexpected.");
  }

  strtok(p, "=; "); /* skip "_nstates" */
  p = strtok((char*)NULL, "=; ");
  nstates = atoi(p);
  printf("nstates = %d\n", nstates);

  /* read _time */
  while(!feof(fpPrb)) {
    fgets(sline, MAXLINE, fpPrb);
    if ((p = strstr(sline, "_time"))) {
      break;
    }
  }

  if (feof(fpPrb)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "read_prb_head reaches EOF unexpected.");
  }

  strtok(p, "=; "); /* skip "_time" */
  p = strtok((char*)NULL, "=; ");
  time0 = atof(p);
  printf("time0 = %f\n", time0);

  /* go to the start of probs */
  while(!feof(fpPrb)) {
    fgets(sline, MAXLINE, fpPrb);
    if ((p = strstr(sline, "_probabilities"))) {
      break;
    }
  }

  if (feof(fpPrb)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "read_prb_head reaches EOF unexpected.");
  }
```

```
}

static int n, nleft = 0;
static double probs[10]; /* to store probs read in one line */

double next_probability() {
  char *p;
  double r;
/*
  printf("next_probability\n");
*/
  if (nleft) {
    r = probs[n - nleft];
    nleft --;
    return r;
  }

  fgets(sline, MAXLINE, fpPrb);
  if (feof(fpPrb)) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "next_probability reaches EOF unexpected.");
  }

  n = 0;
  p = strtok(sline, ": ");
  do {
    if (strlen(p) > 12) { /* a prob. is longer than 12c */
      probs[n] = atof(p);
/*
      printf("prob: %e\n", probs[n]);
*/
      n ++;
    }
    p = strtok((char *)NULL, ": ");
  } while(p);

  nleft = n;

  if (nleft) {
    r = probs[n - nleft];
    nleft --;
    return r;
  }

  return -1.00;
}

void loadprob(char *sfName) {

  int i, j, bFound;
  char sfNameFull[32], sMsg[100];
  static  ME *curmark;

  LogMsg(MSG_PLAIN, "loading init prob. ... ");

  sprintf(sfNameFull, "%s.rg", sfName);
  if ((fpRG = fopen(sfNameFull, "r")) == NULL) {
    fclose(fpRG);
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "'load_init_prob' could not open .rg file.");
```

```
  }

  sprintf(sfNameFull, "%s.prb", sfName);
  if ((fpPrb = fopen(sfNameFull, "r")) == NULL) {
    fclose(fpPrb);
    LogMsg(MSG_EXIT, "'load_init_prob' could not open .prb file.");
  }

  /* read headers of .rg and .prb */
  read_rg_head();
  read_prb_head();

  for(i = 0; i < nstates; i ++) {
/*
    printf("before next_marking.\n");
*/
    next_marking();
/*
    printf("after next_marking.\n");
*/

    printf("\r %d", i);

    for (curmark = FirstCanonic(); curmark ≠ NULL; curmark =
        NextCanonic(curmark)) {

      /* if Prob(i) > 0, then prob of i-th marking is set. */
      if (Prob_init[curmark→index] > 0)
        continue;

      if (curmark→index < 0) continue;

      bFound = 1;
      for(j = 0; j < nplace; j ++) {
        if (curmark→mk[idx[j]] ≠ marking[j]) {
          bFound = 0;
          break;
        }
      }

      if (bFound)
    break;
    }

   /* find the same marking */
   if (bFound) {
      Prob_init[curmark→index] = next_probability();
/*
      printf("%e\n", Prob_init[curmark->index]);
*/
    } else {
      sprintf(sMsg, "'load_init_prob' marking #%d is not found.", i);
      LogMsg(MSG_PLAIN, sMsg);
    }
  }

  fclose(fpRG);
  fclose(fpPrb);

  LogMsg(MSG_PLAIN, "... loading finished.");
}
```

Appendix A

Differences Between last versions of SPNP

A.1 Command Differences Between SPNP Version 4 and Version 5

SPNP Commands	
Version 4	Version 5
parameters(){}	void options(){}
iopt(IOP_METHOD, VAL_SSSOR);	iopt(IOP_SSMETHOD, VAL_SSSOR);
iopt(IOP_METHOD, VAL_TSUNIF);	iopt(IOP_TSMETHOD, VAL_TSUNIF);
rate_type	double
net (){}	void net () {}
assert (){}	int assert (){}
ac_init(){}	void ac_init{}
ac_reach (){}	void ac_reach (){}
reward_type	double
ac_final (){}	void ac_final() {}
no special command	solve(INFINITY)
time_value(time_pt)	solve(time_point)

A.2 Command Differences Between SPNP Version 5 and Version 6

- Functions **ratenoval** and **probnoval** have been removed. **rateval(char *t, 1.0)** and **probval(char *t, 1.0)** must be used instead.
- The FSPNs, the discrete event simulator and the importance splitting methods have been added.

Appendix B

SPNP Applications

In this section, we list some papers in which SPNP was used. The list does not include all the paper where SPNP was used.

1. G. Agrawal, "Availability of Coding Based Replication Schemes," *Eleventh IEEE Symposium on Reliable Distributed Systems*, Houston, 1992.

2. M. Balakrishnan, A. Puliafito, K. S. Trivedi and I. Viniotis, "Buffer Sizing for Available Bit Rate (ABR) Traffic in an ATM Switch." *IEEE International Conference on Communications*, 1995. Full version submitted to the *J. of Telecomm. Systems, Baltzer Science Publishers, Zurich.*

3. M. Balakrishnan and K. S. Trivedi, "Stochastic Petri Nets for the Reliability Analysis of Communication Network Applications with Alternate-Routing". *Reliability Engineering and System Safety*, special issue on Reliability and Safety Analysis of Dynamic Process Systems, Vol. 52, No. 3, pp. 243–259, 1996.

4. I.R. Chen and R. Betapudi, "A Petri net model for the performance analysis of transaction database systems with continuous deadlock detection." *1994 ACM/SIGAPP Symp. on Applied Computing (SAC '94)*, March, 1994.

5. I. R. Chen, and T. W. Tsao, "A reliability model for real-time expert systems". *IEEE Transactions on Reliability*, Dec. 1994.

6. H. Choi and K. S. Trivedi, "Approximate Performance Models of Polling Systems using Stochastic Petri Nets." *Proceedings of the IEEE INFOCOM 92*, Florence,Italy, May 4-8,1992.

7. G. Ciardo and K. S. Trivedi, "Solution of Large Generalized Stochastic Petri Net Models," in: *Numerical Solution of Markov Chains,* W. J. Stewart (ed.), Marcel Dekker, New York, 1991.

8. G. Ciardo, J. Muppala, and K. S. Trivedi, "Analyzing Concurrent and Fault-Tolerant Software using Stochastic Reward Nets." *Journal of Parallel and Distributed Computing*, 15:255-269, 1992.

9. G. Ciardo and K. S. Trivedi, "A Decomposition Approach for Stochastic Petri Net Models," *International Conference on Petri Nets and Performance Models,* Melbourne, Australia, December 1991, also *Performance Evaluation*, Vol. 18, No. 1, pp. 37-59, July 1993.

10. G. Ciardo, L. Cherkasova, V. Kotov and T. Rokicki, "Modeling a Scalable High-Speed Interconnect with Stochastic Petri Nets," *Proceedings of the 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95),* Durham, NC, USA, Oct. 1995.

11. C. Constantinescu and K. S. Trivedi, "Dependability Modeling of Real-Time Systems Using Stochastic Reward Nets," *Microelectronics and Reliability,* Vol. 35, No. 6, pp. 903-914, 1995.

12. R. Fricks, C. Hirel, S. Wells and K. S. Trivedi, "The Development of an Integrated Modeling Environment," *Proceedings of the 1st World Congress on Systems Simulation (WCSS'97,* Singapore, Sept. 1997.

13. S. Greiner, A. Puliafito, G. Bolch and K. S. Trivedi, "Performance Evaluation of Dynamic Priority Operating Systems," *Proceedings of the 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95),* Durham, NC, USA, Oct. 1995.

14. C. Hirel, S. Wells, R. Fricks and K. S. Trivedi, "iSPN: an Integrated Environment for Modeling Using Stochastic Petri Nets, Tools Demonstration," *Joint Conference PNPM'97 and PERFORMANCE'97,* Saint-Malo, France, June 1997.

15. S. Hunter, T. Phillip and K. S. Trivedi, "Combined Performance and Availability Analysis of a Switched Network Application", *IEEE International Conference on Communications (ICC'97),* Montréal, Québec, Canada, 8-12 June, 1997.

16. O. C. Ibe and K. S. Trivedi, "Stochastic Petri Net Models of Polling Systems," *IEEE Journal on Selected Areas in Communications,* Vol. 8, No. 9, pp. 1649-1657, Dec. 1990.

17. O. C. Ibe, K. S. Trivedi, A. Sathaye, and R. C. Howe, "Stochastic petri net modeling of vaxcluster system availability," in *Proceedings of the International Conference on Petri Nets and Performance Models,* (Kyoto, Japan), December 1989.

18. O. C. Ibe and K. S. Trivedi, "Stochastic Petri Net Analysis of Finite-Population Vacation Queueing Systems," *Queueing Systems: Theory and Applications,* Vol. 8, No. 2, pp. 111-128, 1991.

19. O. C. Ibe, H. Choi, and K. S. Trivedi. "Performance Evaluation of Client-Server Systems", *IEEE Transactions on Parallel and Distributed Systems*. Vol. 4, No. 11, November 1993, pp. 1217-1229.

20. F. J. Jaimes-Romero, D. Muñoz-Rodríguez, C. Molina and H. Tawfik, "Modeling Recource Management in Cellular Systems Using Petri Nets", *IEEE Trans. Veh. Technol.*, Vol. 46, No. 22, pp. 298-312, May, 1997.

21. H. Kantz and K. S. Trivedi, "Reliability Modeling of the MARS System: A Case Study in the Use of Different Tools and Techniques," *International Conference on Petri Nets and Performance Models,* Melbourne, Australia, December 1991.

22. N. Lopez-Benitez, "Dependability Analysis of Distributed Computing Systems Using Stochastic Petri Nets," *Eleventh IEEE Symposium on Reliable Distributed Systems,* Houston, 1992.

23. Y. Ma, J. J. Han and K. S. Trivedi, " A Channel Recovery Method in TDMA Wireless Systems", *Proc. of 50th IEEE International Vehicular Technology Conference (VTC Fall'99),* Amsterdam, The Netherlands, Sep. 1999.

24. Y. Ma, J. J. Han and K. S. Trivedi, "A Channel Recovery Method for RF Channel Failure in Wireless Communications Systems", *Proc. of IEEE Wireless Communications and Networking Conference (WCNC'99),* New Orleans, LA, Sep. 1999.

25. Y. Ma, C. W. Ro and K. S. Trivedi. "Performability Analysis of Channel Allocation with Channel Recovery Strategy in Cellular Networks". To appear in Proceedings of the 7th IEEE International Conference on Universal Personal Communications (ICUPC'98), Florence, Italy, 5-9 October, 1998.

26. M. Madhukar, M. Leuze, and L. Dowdy, "Petri Net Model of a Dynamically Partitioned Multiprocessor System," *Proceedings of the 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95),* Durham, NC, USA, Oct. 1995.

27. C. Molina, N. Jain and K. Basu. "Performance Model of Cellular Data on American Systems". In Proceedings of 1996 IEEE 46th Vehicular Technology Conference (VTC'96). Atlanta, Georgia, USA, 28 April - 1 May, 1996.

28. J. K. Muppala, and K. S. Trivedi. "Composite performance and availability analysis using a hierarchy of stochastic reward nets." In G. Balbo (Ed.), *Proc. Fifth Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*. Torino, Italy, 1991, 322–336.

29. J. K. Muppala, S. P. Woolet, and K. S. Trivedi. " Real-time systems performance in the presence of failures", *IEEE Computer*, May 1991.

30. J. K. Muppala, A. Sathaye, R. Howe and K. S. Trivedi. "Dependability Modeling of a Heterogeneous VAXcluster System Using Stochastic Reward Nets," in: *Hardware and Software Fault Tolerance in Parallel Computing Systems,* D. Avresky (ed.), Ellis Horwood Ltd., pp.33-59, 1992.

31. J. K. Muppala, and K. S. Trivedi. "Numerical transient analysis of finite Markovian queueing systems." In Basawa and Bhat (Eds.), *Queueing and Related Models*, Oxford University Press, 1992, 262-284.

32. J. K. Muppala, S.P. Woolet, and K.S. Trivedi, "On Modeling Performance of Real-Time Systems in the Presence of Failures," in: *Readings in Real-Time Systems,* Y.-H. Lee and C. M. Krishna (eds.), pp. 219-239, IEEE Press, 1993.

33. J. K. Muppala, Varsha Mainkar, Vidyadhar Kulkarni and K. S. Trivedi. Numerical computation of response time distributions using stochastic reward nets. *Annals of Operations Research*. Vol 48, pp. 155-184, 1994.

34. J. K. Muppala, G. F. Ciardo and K. S. Trivedi. "Stochastic reward nets for reliability prediction." *Communications in Reliability, Maintainability and Serviceability: An International Journal published by SAE International,* Vol. 1, No. 2, pp. 9-20, July 1994.

35. C. W. Ro and K. S. Trivedi. "Performability Analysis of Handoff Calls in Personal Communication Networks," *Proc. Sixth International Conference on Computer Communications and Networks (IC3N'97),* Las Vegas, Nevada, Sept. 1997.

36. H. Sun, X. Zang and K.S. Trivedi. "A stochastic reward net model for performance analysis of prioritized DQDB MAN", *Computer Communications*, Vol.22, No. 9, pp. 858-870, June 1999.

37. H. Sun, X. Zang. and K.S. Trivedi. "Performance of Broadcast and Unknown Server (BUS) in ATM LAN Emulation", Technical Report. *Center for Advanced Computing and Communication, Duke University*, 1999.

38. H. Sun, X. Zang. and K.S. Trivedi. "A Performance Model of Partial Packet Discard and Early Packet Discard Schemes in ATM Switches", Technical Report. *Center for Advanced Computing and Communication, Duke University*, 1999.

39. L. Tomek and K. S. Trivedi, "Fixed-Point Iteration in Availability Modeling," in: *Informatik-Fachberichte, Vol. 91: Fehlertolerierende Rechensysteme,* M. Dal Cin (ed.), Springer-Verlag, Berlin, 1991.

40. L. Tomek, J. K. Muppala and K. S. Trivedi. "Modeling correlation in software recovery blocks." *IEEE Transactions on Software Engineering, Special Issue on Software Reliability*, 19(11), November 1993.

41. L. Tomek, V. Mainkar, R. Geist and K. Trivedi. "Reliability analysis of life-critical real-time systems." *Proceedings of the IEEE*, January 1994.

42. L. Tomek and K.S. Trivedi. "Analyses Using Stochastic Reward Nets", in: *Software Fault Tolerance,* M. Lyu (ed.), John Wiley & Sons, 1994.

43. K. S. Trivedi, Y. Ma and J. J. Han, "Performability analysis of fault tolerant RF link design in wireless communications networks", invited paper in *Proc. of the 13th European Simulation Multiconference (ESM99),* Warsaw, Poland, June 1999.

44. K. S. Trivedi, S. Hunter, S. Garg and R. M. Fricks. "Reliability Analysis Techniques Explored Through a Communication Network Example." in *Proceedings of the Computer-Aided Design, Test, and Evaluation for Dependability Conference: CADTED'96*, Beijing, China, July 1996, pp. 110-120.

45. B. Tuffin, D.S. Chen and K.S. Trivedi. "Comparison of Hybrid Systems and Fluid Stochastic Petri Nets", Technical Report. *Center for Advanced Computing and Communication, Duke University,* 1999.

46. C.-Y. Wang, D. Logothetis, I. Viniotis and K. S. Trivedi, "Transient Behavior of ATM Networks under Overloads," *Proceedings of the IEEE INFOCOM 96*, San Francisco, CA, pp. 978-985, March 1996.

Bibliography

[1] M. Ajmone-Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[2] J. T. Blake, A. L. Reibman, and K. S. Trivedi. Sensitivity analysis of reliability and performance measures for multiprocessor systems. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Fe, U.S.A., May, 1988.

[3] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains, Modeling and Performance Evaluation with Computer Science Application*. Wiley & Sons, 1998.

[4] H. Choi and K. S. Trivedi. Approximate performance models of polling systems using stochastic Petri nets. In *Proceedings of IEEE Infocom 92, 11th Annual Joint Conference of the IEEE Computer and Communication Societies*, Florence Italy, May 1992.

[5] G. Ciardo, A. Blakemore, Jr. P. F. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using stochastic reward nets. In C. Meyer and R. Plemmons, editors, *Linear Algebra, Markov Chains and Queuing Models*, volume 48, pages 145–191. Springer-Verlag, 1993.

[6] G. Ciardo, J. K. Muppala, and K. S. Trivedi. SPNP: Stochastic Petri Net Package. In *Proceedings of 3rd International Workshop on Petri Nets and Performance Models*, pages 142–150, Kyoto, Japan, Dec. 1989.

[7] G. Ciardo, J. K. Muppala, and K. S. Trivedi. On the solution of GSPN reward models. *Performance Evaluation*, 12(4):237–254, July, 1991.

[8] G. Ciardo, D.M. Nicol, and K.S. Trivedi. Discrete-Event Simulation of Fluid Stochastic Petri-Nets. *IEEE Transactions on Software Engineering*, 25(2):207–217, 1999.

[9] G. Ciardo and K. S. Trivedi. Solution of large generalized stochastic petri net models. In W. J. Stewart, editor, *Numerical Solution of Markov Chains*. Marcel Dekker, 1991.

[10] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[11] Computer Science Department. College of Willian and Mary. *On the Simulation of Stochastic Petri Nets*.

[12] J. B. Dugan, K. S. Trivedi, R. M. Geist, and V. F. Nicola. Extended stochastic Petri nets: Applications and analysis. In E. Gelenbe, editor, *Performance '84*, pages 507–519. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, Netherlands, 1985.

[13] S. P. Harbison and G. L. Steele Jr. *C — A Reference Manual*. Prentice-Hall, 3 edition, 1991.

[14] G. Horton, V. Kulkarni, D. Nicol, and K.S. Trivedi. Fluid Stochastic Petri nets: Theory, Application and Solution. *European Journal of Operational Research*, 105:184–201, 1998.

[15] R. A. Howard. *Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Process.* John Wiley and Sons, New York, NY, USA, 1971.

[16] H. Kantz and K. S. Trivedi. Reliability modeling of the mars system: A case study in the use of different tools and techniques. In *International Conference on Petri Nets and Performance Models*, Melbourne, Australia, Dec. 1991.

[17] Y. Ma, C. W. Ro, and K. S. Trivedi. Performability analysis of channel allocation with channel recovery strategy in cellular networks. In *Proceedings of the 7th IEEE International Conference on Universal Personal Communications (ICUPC'98)*, Florence, Italy, October 1998.

[18] M. Malhotra, J. K. Muppala, and K. S. Trivedi. Stiffness-tolerant methods for transient analysis of stiff Markov chains. *Microelectron. Relib.*, 34(11):1825–1841, 1994.

[19] J. K. Muppala and K. S. Trivedi. Composite Performance and Availability Analysis using a Hierarchy of Stochastic Reward Nets. In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation, Modelling Techniques and Tools*, pages 335–350. Elsevier, Amsterdam, 1992.

[20] J. K. Muppala and K. S. Trivedi. Numerical transient analysis of finite markovian queueing systems. In U. N. Bhat and I. V. Basawa, editors, *Queueing and Related Models*, pages 262–284. Oxford University Press, 1992.

[21] J. K. Muppala and K. S. Trivedi. GSPN models: Sensitivity analysis and applications. In *Proceedings of the 28th ACM Southeast Region Conference*, pages 24–33, Apr. 1990.

[22] J. K. Muppala, S. P. Woolet, and K. S. Trivedi. Real-time performance in the presence of failures. *IEEE Computer*, May 1991.

[23] I. Mura, A. Bondavalli, X. Zang, and K. S. Trivedi. Dependability modelling and evaluation of phased mission systems: a DSPN approach. In *10th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation (Performance Tools'98)*, Palma de Mallorca, Spain, Sep. 1998, Submitted.

[24] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April,1989.

[25] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, 1981.

[26] A. L. Reibman, R. M. Smith, and K. S. Trivedi. Markov and Markov reward model transient analysis: An overview of numerical approaches. *European Journal of Operational Research*, 40:257–267, 1989.

[27] R. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using SHARPE Software Package*. Kluwer Academic Publishers, 1995.

[28] R. M. Smith, K. S. Trivedi, and A. V. Ramesh. Performability analysis: measures, an algorithm, and a case study. *IEEE Trans. Comput.*, 37(4):406–417, Apr. 1988.

[29] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[30] L. Tomek and K. S. Trivedi. Informatik-fachberichte, vol. 91: Fehlertolerierende rechensysteme. In M. Dal Cin, editor, *Fixed-Point Iteration in Availability Modeling*. Springer-Verlag, Berlin, 1991.

[31] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Inc., Englewood Cliffs, 1982.

[32] K. S. Trivedi and V. G. Kulkarni. FSPNs: Fluid Stochistic Petri Nets. In *14th International Conference on Applacations and Theory of Petri Nets*, pages 24–31, 1993.

[33] K. S. Trivedi, J. K. Muppala, S. P. Woolet, and B. R. Haverkort. Composite performance and dependability analysis. *Performance Evaluation*, 14(3-4):197–215, Feb. 1992.

[34] B. Tuffin and K.S. Trivedi. Implementation of importance splitting techniques in stochastic petri net package. Technical report, Duke University, Durham, NC, 1999.

[35] W. B. van den Hout. *The Power-Series Algorithm: A Numerical Approach to Markov Processes*. Tilburg University, 1996.

Index

absorbing marking, 4
ac_final, 29
ac_init, 29
ac_reach, 29
accumulated, 39
affected, 13
arc, 3
assert, 28

beta distribution, 25
betdep, 25
betfun, 25
betval, 25
bind, 19
binodep, 25
binofun, 25
binomial distribution, 26
binoval, 25

Cauchy distribution, 27
caudep, 27
caudep_is, 48
caufun, 27
caufun_is, 49
cauval, 27
cauval_is, 48
Creation of the SRN in iSPN, 69
CSPL, 1, 7

detdep, 23
deterministic transtions, 23
detfun, 23
detval, 23
diarc, 22
dmharc, 22
dmiarc, 22
dmoarc, 22
dharc, 22
doarc, 22
dvharc, 22
dviarc, 22
dvoarc, 22

enable, 3
enabled, 16
ergodic, 39
Erlang distribution, 27
erldep, 27
erldep_is, 48
erlfun, 27
erlfun_is, 48
erlval, 27
erlval_is, 48
Execution of a model in iSPN, 75
expected, 32

fbound, 21
fbreak, 21
fcondition, 23
fiarc, 21
File functions in iSPN, 72
finit, 21
finput, 11
texbffinput, 10
fire, 3
fliarc, 22
floarc, 22
fmark, 22
fmiarc, 21
fmoarc, 21
foarc, 21
fopt, 10
fplace, 20
fviarc, 21
fvoarc, 21

gamdep, 25
gamfun, 25
gamma distribution, 25
gamval, 25
geomdep, 23
geometric distribution, 24

geomfun, 23
geomval, 23
guard, 4
guard, 17

halting_condition, 14
harc, 13
hold_cond, 41
hyperdep, 26
hyperdep_is, 48
hyperexponential distribution, 26
hyperfun, 26
hyperfun_is, 48
hyperval, 26
hyperval_is, 48
hypo-exponential distribution, 27
hypodep, 26
hypofun, 26
hypoval, 26

iarc, 13
imm, 12
immediate transition, 4, 12
importance sampling, 47
importance splitting, 46
inf, 21
inhibitor arc, 4, 14
init, 12
input, 11
texbfinput, 10
input arc, 3, 14
iopt, 10

logndep, 24
lognfun, 24
lognormal distribution, 24
lognval, 24
loop, 5

mark, 16
marking dependent, 4, 5, 16, 17
mharc, 14
miarc, 14
moarc, 14
Modification of the SRN in iSPN, 70
MRM, 1
multiplicity, 3, 14

negative binomial distribution, 26
negbdep, 26
negbfun, 26
negbval, 26
net, 11
non-null recurrent, 39
normal distribution, 24
normdep, 24
normfun, 24
normval, 24
null recurrent, 39

oarc, 13
options , 9
output arc, 3, 14

pardep, 27
pardep_is, 48
Pareta distribution, 27
parfun, 27
parfun_is, 48
parm, 19
parval, 27
parval_is, 48
Petri net, 3
place, 3, 12
place, 12
poisdep, 25
poisfun, 25
Poisson distribution, 25
poisval, 25
policy, 13
positive recurrent, 39
pr_accumulated, 39
pr_cum_abs, 34
pr_cum_expected, 33, 45
pr_expected, 31, 45
pr_hold_cond, 41
pr_mc_info, 31
pr_message, 35

pr_mtta, 34
pr_mtta_fun, 34
pr_net_info, 29
pr_parms, 29
pr_rg_info, 29
pr_std_average, 31
pr_std_cum_average, 33
pr_time_avg_expected, 33
pr_value, 35
priority, 4
priority, 13
probdep, 18
probdep_is, 49
probfun, 18
probfun_is, 49
probnoval, 19
probval, 12, 49

rate, 16
rate dependent, 16
ratedep, 18
ratedep_is, 48
ratefun, 18
ratefun_is, 48
ratenoval, 19
rateval, 12
rateval_is, 48
reachability set, 4
reachable, 4
recurrent, 39
regenerative simulation, 49
regenerative simulation with importance sampling, 49
resampling, 49
RESTART, 46
reward function, 32

sensitivity analysis, 19
set_prob0, 40
set_prob_init, 34
simulation, 43
solve, 30
splitting, 46
splitting, 47
SRN, 1
steady state analysis, 30
stochastic Petri net, 4

tangible marking, 4
timed transition, 4, 12
transient, 39
transient analysis, 30
transition, 3

unifdep, 23
unifdep_is, 48
uniffun, 23
uniffun_is, 48
uniform distribution, 23
unifval, 23
unifval_is, 48
useparm, 19

vanishing marking, 4
vharc, 18
viarc, 18
voarc, 18

weibdep, 24
weibdep_is, 48
weibfun, 24
weibfun_is, 48
Weibull distribution, 24
weibval, 24
weibval_is, 48